May 2004

Typed Intermediate Languages

Stephen Tse
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-17.

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
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-04-17.

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.

Stephen Tse (`stse@cis.upenn.edu`)
Survey report for PhD qualifying exam (WPE-II).
Last update: May 25, 2004.

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].

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 term 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 bad 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 $\mathtt{f(x)} = \mathtt{x} \times \mathtt{f(x} - \mathtt{1)}$ with $\mathtt{f(0)} = \mathtt{1}$. 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 $\mathtt{S}$ and the register file $\mathtt{R}$ as follows:

$$
\begin{array}{rcl}
\mathtt{iconst\ n} & : & \mathtt{S} \# \mathtt{R} \longrightarrow \mathtt{S}, \mathtt{n} \# \mathtt{R} \\
\mathtt{iload\ r} & : & \mathtt{S} \# \mathtt{R}, \mathtt{r} \mapsto \mathtt{n} \longrightarrow \mathtt{S}, \mathtt{n} \# \mathtt{R}, \mathtt{r} \mapsto \mathtt{n} \\
\mathtt{isub} & : & \mathtt{S}, \mathtt{n}_1, \mathtt{n}_2 \# \mathtt{R} \longrightarrow \mathtt{S}, \mathtt{n}_1 - \mathtt{n}_2 \# \mathtt{R} \\
\mathtt{imul} & : & \mathtt{S}, \mathtt{n}_1, \mathtt{n}_2 \# \mathtt{R} \longrightarrow \mathtt{S}, \mathtt{n}_1 \times \mathtt{n}_2 \# \mathtt{R} \\
\mathtt{invokestatic\ <t\ m(t_1, \ldots, t_n)>} & : & \mathtt{S}, \mathtt{v}_1, \ldots, \mathtt{v}_n \# \mathtt{R} \longrightarrow \mathtt{S}, \mathtt{v} \# \mathtt{R} \quad \mathtt{if} \quad \mathtt{v} = \mathtt{m}(\mathtt{v}_1, \ldots, \mathtt{v}_n)
\end{array}
$$

Here $\mathtt{S}$ is a linear list of values and we use $\mathtt{S}, \mathtt{n}$ to represent the new stack of $\mathtt{S}$ with the value $\mathtt{n}$ pushed on top. $\mathtt{R}$ is finite mapping from indices to values and we use $\mathtt{R}, \mathtt{r} \mapsto \mathtt{n}$ to specify that $\mathtt{r}$ is a valid register of value $\mathtt{n}$. 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:

$$
\mathtt{iconst\ n} \ : \ \mathtt{S} \# \mathtt{R} \longrightarrow \mathtt{S}, \mathtt{int} \# \mathtt{R}
$$

$$
\begin{array}{rcl}
\texttt{iload r} & : & \texttt{S} \# \texttt{R}, \texttt{r:int} \longrightarrow \texttt{S}, \texttt{int} \# \texttt{R}, \texttt{r:int} \\
\texttt{isub} & : & \texttt{S}, \texttt{int}, \texttt{int} \# \texttt{R} \longrightarrow \texttt{S}, \texttt{int} \# \texttt{R} \\
\texttt{imul} & : & \texttt{S}, \texttt{int}, \texttt{int} \# \texttt{R} \longrightarrow \texttt{S}, \texttt{int} \# \texttt{R} \\
\texttt{invokestatic <t m}(\texttt{t}_1, \ldots, \texttt{t}_n)\texttt{>} & : & \texttt{S}, \texttt{t}'_1, \ldots, \texttt{t}'_n \# \texttt{R} \longrightarrow \texttt{S}, \texttt{t} \# \texttt{R} \quad \texttt{if} \quad \texttt{t}'_1 \texttt{<:} \texttt{t}_1, \ldots, \texttt{t}'_n \texttt{<:} \texttt{t}_n
\end{array}
$$

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 ($\texttt{<t m}(\texttt{t}_1, \ldots, \texttt{t}_n)\texttt{>}$) without executing the method.

The rules above specify the following safety properties of execution: (1) for `iload r`, register `r` must be an integer ($\texttt{R}, \texttt{r} : \texttt{int}$), (2) for `isub` and `imul`, the top two operands must both be integers ($\texttt{S}, \texttt{int}, \texttt{int}$), and (3) for `invokestatic`, the actual argument types ($\texttt{t}'_1, \ldots, \texttt{t}'_n$) from the operand stack must be subtypes ($\texttt{t}'_1 \texttt{<:} \texttt{t}_1, \ldots, \texttt{t}'_n \texttt{<:} \texttt{t}_n$) of the formal argument types ($\texttt{t}_1, \ldots, \texttt{t}_n$) 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 $\epsilon \# \texttt{t}_1, \ldots, \texttt{t}_n, \top, \ldots, \top$, is an empty stack ($\epsilon$) and register file with types of method parameters ($\texttt{t}_1, \ldots, \texttt{t}_n$) and with the top type $\top$ 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 $\texttt{C}_1$ and $\texttt{C}_2$ extend $\texttt{C}_0$, and if a register from one predecessor instruction has type $\texttt{C}_1$ and that from another has $\texttt{C}_2$, then the merged output state has type $\texttt{C}_0 = \texttt{lub}(\texttt{C}_1, \texttt{C}_2)$ 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 $\texttt{I}(\ell)$ for the instruction at program location ℓ, write $\texttt{IN}(\ell)$ for the input state of the instruction at ℓ, and write $\texttt{OUT}(\ell)$ for the output state. In particular, $\texttt{IN}(0)$ specifies the input state at the beginning of the method. The dataflow equations for bytecode verification are:

$$\texttt{I}(\ell) \quad : \quad \texttt{IN}(\ell) \longrightarrow \texttt{OUT}(\ell)$$

```
class Main {
  static void f () {
    int x, y;
    try {
      x = 0;
    } finally {}
    y = x;
  }
}
```

```
Method void f() with [(0,4,Exception,8)]
   0 iconst_0     %  0
   1 istore_0     %  x=0
   2 jsr 14
   5 goto 17
   8 astore_2     %  catch (Exception e)
   9 jsr 14
  12 aload_2      %  Exception e
  13 athrow       %  throw e
  14 astore_3     %  return address = 5
  15 ret 3
  17 iload_0      %  x
  18 istore_1     %  y = x
  19 return
```

Figure 2: Exception handling in Java source and bytecode

$$\begin{aligned} \mathtt{IN}(\ell) &= \mathtt{lub}\{\mathtt{OUT}(\ell') \mid \ell' \in \mathtt{PRED}(\ell)\} \\ \mathtt{IN}(0) &= \epsilon \# \mathtt{t}_1, \ldots, \mathtt{t}_n, \top, \ldots, \top \end{aligned}$$

where $\mathtt{PRED}(\ell)$ is the set of predecessors of instruction at ℓ. We can readily compute $\mathtt{PRED}(\ell)$ 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 `jsr` (jump to subroutine) and `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 `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 `try` block, the implicit `catch` block, the `finally` block, and the last statement in the Java source. The key point is to illustrate that the `try` block and the `catch` block share the same subroutine for the `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 `try` block, while those at 14-15 correspond to the `finally` block. The instruction `jsr 14` pushes the return address, which is 5 in this example, to the stack and jumps to 14. Inside the `finally` block, the return address is first saved to register 3 so that the instruction `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, $\mathtt{[(0,4,Exception,8)]}$ 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 ($\mathtt{R(0)} = \mathtt{int}$) but not in the `catch` block ($\mathtt{R(0)} = \top$) and thus their merged type is uninitialized ($\mathtt{R(0)} = \top$). 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 $\mathtt{y} = \mathtt{x}$ does not pass the verification because $\mathtt{R(0)} = \top$.

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 $\mathtt{S} \mathbin{\#} \mathtt{R}$ at each program location is now parameterized by the subroutine call stack `C`, which is also called a *contour*. A dataflow equation at ℓ becomes $\mathtt{I(\ell) : IN(\ell;\ C)} \longrightarrow \mathtt{OUT(\ell;\ C')}$.

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

More formally, for the instruction $\mathtt{jsr}\ \ell$ at ℓ_1 followed by an instruction at ℓ_2 in the call stack `C`, the equation is:

$$\mathtt{IN}(\ell;\ \mathtt{C}, \ell_2) = \mathtt{S}, \mathtt{RA}(\ell_2) \mathbin{\#} \mathtt{R} \qquad \mathtt{if} \quad \mathtt{OUT}(\ell_1;\ \mathtt{C}) = \mathtt{S} \mathbin{\#} \mathtt{R}$$

The equation says that the analysis pushes ℓ_2 to `C` and pushes $\mathtt{RA}(\ell_2)$ to the operand stack `S`, where $\mathtt{RA}(\ell_2)$ is the type representing the return address at ℓ_2. Note that $\mathtt{RA}(\ell_2)$ 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 ℓ_1 in `C`, the equation is:

$$\mathtt{IN}(\ell_2;\ \mathtt{C}) = \mathtt{OUT}(\ell_1;\ \mathtt{C}', \ell_2, \mathtt{C}) = \mathtt{S} \mathbin{\#} \mathtt{R}, \mathtt{n}{:}\mathtt{RA}(\ell_2)$$

The equation says that, from the return address ℓ_2 at register `n`, the analysis pops the call stack $\mathtt{C}', \ell_2, \mathtt{C}$ at ℓ_1 until the address ℓ_2 to get `C`, and then propagate the output state at ℓ_1 back to the input state at ℓ_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 $\ell \mid \mathtt{S} \;\#\; \mathtt{R} \longrightarrow \ell' \mid \mathtt{S}' \;\#\; \mathtt{R}'$ for the transition function $\mathtt{I}(\ell) : \mathtt{S} \;\#\; \mathtt{R} \longrightarrow \mathtt{S}' \;\#\; \mathtt{R}'$ in Section 2.1:

$$
\begin{array}{lcll}
\ell \mid \mathtt{S} \;\#\; \mathtt{R} & \longrightarrow & \ell' \mid \mathtt{S}',\mathtt{R}' & \text{if} \;\; \mathtt{I}(\ell) : \mathtt{S} \;\#\; \mathtt{R} \longrightarrow \mathtt{S}' \;\#\; \mathtt{R}' \quad \text{and} \quad \ell' \in \mathtt{SUCC}(\ell) \\
\ell \mid \mathtt{S} \;\#\; \mathtt{R} & \longrightarrow & \bot & \text{if} \;\; \mathtt{I}(\ell) : \mathtt{S} \;\#\; \mathtt{R} \not\longrightarrow
\end{array}
$$

where $\mathtt{SUCC}(\ell)$ is the set of successors of instruction at ℓ 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 $0 \mid \epsilon \;\#\; \mathtt{t}_1, \ldots, \mathtt{t}_n, \top, \ldots, \top$ (similar to the $\mathtt{IN}(0)$ in Section 2.2). The verification succeeds if the closure of reachable states does not include $\bot$.

For instance, the successor relation for instructions `jsr` and `ret` are:

$$
\begin{array}{lcll}
\ell \mid \mathtt{S} \;\#\; \mathtt{R} & \longrightarrow & \ell' \mid \mathtt{S}, \mathtt{RA}(\ell+3) \;\#\; \mathtt{R} & \text{if} \;\; \mathtt{I}(\ell) = \mathtt{jsr}\ \ell' \\
\ell \mid \mathtt{S} \;\#\; \mathtt{R} & \longrightarrow & \ell' \mid \mathtt{S} \;\#\; \mathtt{R} & \text{if} \;\; \mathtt{I}(\ell) = \mathtt{ret\ r} \quad \text{and} \quad \mathtt{R}(\mathtt{r}) = \mathtt{RA}(\ell') \\
\ell \mid \mathtt{S} \;\#\; \mathtt{R} & \longrightarrow & \bot & \text{if} \;\; \mathtt{I}(\ell) = \mathtt{ret\ r} \quad \text{and} \quad \mathtt{R}(\mathtt{r}) \neq \mathtt{RA}(\ell')
\end{array}
$$

because the size of instruction $\mathtt{jsr}\ \ell'$ 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 $\mathtt{y} = \mathtt{x}$ at 18 will not reach $\bot$.

$$
\begin{array}{rcl}
2 \mid \epsilon \;\#\; 0{:}\mathtt{int} & \longrightarrow & 14 \mid \mathtt{RA}(5) \;\#\; 0{:}\mathtt{int} \\
9 \mid \epsilon \;\#\; 2{:}\mathtt{Exception} & \longrightarrow & 14 \mid \mathtt{RA}(12) \;\#\; 2{:}\mathtt{Exception} \\
14 \mid \epsilon \;\#\; 0{:}\mathtt{int}, 3{:}\mathtt{RA}(5) & \longrightarrow & 5 \mid \epsilon \;\#\; 0{:}\mathtt{int}, 3{:}\mathtt{RA}(5) \\
14 \mid \epsilon \;\#\; 2{:}\mathtt{Exception}, 3{:}\mathtt{RA}(12) & \longrightarrow & 12 \mid \epsilon \;\#\; 2{:}\mathtt{Exception}, 3{:}\mathtt{RA}(12)
\end{array}
$$

Note that the number of reachable states are finite since the number of program locations ℓ, 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, $C^\sharp$ 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-

```
prod:   mov r3, 0;         % initialize result
        jmp loop           %
loop:   bz r1, done        % branch if r1 == 0
        add r3, r3, r2     % r3 = r3 + r2
        add r1, r1, -1     % r1 = r1 - 1
        jmp loop           %
done:   jmp r4             % return
```

Figure 3: Product function in TAL-0

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 programming [12, 54, 87]. On the other hand, many JVM instructions are very complex and prohibit optimizations at the bytecode level: the instruction `invokevirtual` in Java bytecode 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 environment 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:

$$
\begin{array}{lll}
\text{Operands} & \mathtt{v} & ::= \ \mathtt{r} \mid \mathtt{n} \mid \ell \\
\text{Instructions} & \mathtt{i} & ::= \ \mathtt{mov}\ \mathtt{r},\mathtt{v} \mid \mathtt{add}\ \mathtt{r},\mathtt{r},\mathtt{v} \mid \mathtt{bz}\ \mathtt{r},\mathtt{v} \\
\text{Blocks} & \mathtt{I} & ::= \ \mathtt{jmp}\ \mathtt{v} \mid \mathtt{i};\ \mathtt{I} \\
\text{Heap} & \mathtt{H} & ::= \ \epsilon \mid \mathtt{H},\ell \mapsto \mathtt{I} \\
\text{Register file} & \mathtt{R} & ::= \ \epsilon \mid \mathtt{R},\mathtt{r} \mapsto \mathtt{n} \mid \mathtt{R},\mathtt{r} \mapsto \ell
\end{array}
$$

To simplify the presentation, only three instructions are supported: (1) `mov r v` moves

$$
\begin{array}{rcllr}
\mathtt{jmp}\ \ell \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{H}(\ell) \mid \mathtt{H} \# \mathtt{R} & & \text{(E-JmpL)} \\
\mathtt{jmp}\ \mathtt{r} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{H}(\mathtt{R}(\mathtt{r})) \mid \mathtt{H} \# \mathtt{R} & & \text{(E-JmpR)} \\
\mathtt{mov}\ \mathtt{r}_1, \ell;\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{I} \mid \mathtt{H} \# \mathtt{R}, \mathtt{r}_1 \mapsto \ell & & \text{(E-MovL)} \\
\mathtt{mov}\ \mathtt{r}_1, \mathtt{n};\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{I} \mid \mathtt{H} \# \mathtt{R}, \mathtt{r}_1 \mapsto \mathtt{n} & & \text{(E-MovN)} \\
\mathtt{mov}\ \mathtt{r}_1, \mathtt{r}_2;\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{I} \mid \mathtt{H} \# \mathtt{R}, \mathtt{r}_1 \mapsto \mathtt{R}(\mathtt{r}_2) & & \text{(E-MovR)} \\
\mathtt{add}\ \mathtt{r}_1, \mathtt{r}_2, \mathtt{n};\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{I} \mid \mathtt{H} \# \mathtt{R}, \mathtt{r}_1 \mapsto \mathtt{R}(\mathtt{r}_2) + \mathtt{n} & & \text{(E-AddN)} \\
\mathtt{add}\ \mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_3;\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{I} \mid \mathtt{H} \# \mathtt{R}, \mathtt{r}_1 \mapsto \mathtt{R}(\mathtt{r}_2) + \mathtt{R}(\mathtt{r}_3) & & \text{(E-AddR)} \\
\mathtt{bz}\ \mathtt{r}_1, \ell;\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{H}(\ell) \mid \mathtt{H} \# \mathtt{R} & \text{if} \quad \mathtt{R}(\mathtt{r}_1) = 0 & \text{(E-BzL)} \\
\mathtt{bz}\ \mathtt{r}_1, \mathtt{r}_2;\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{H}(\mathtt{R}(\mathtt{r}_2)) \mid \mathtt{H} \# \mathtt{R} & \text{if} \quad \mathtt{R}(\mathtt{r}_1) = 0 & \text{(E-BzR)} \\
\mathtt{bz}\ \mathtt{r}_1, \mathtt{v};\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \longrightarrow & \mathtt{I} \mid \mathtt{H} \# \mathtt{R} & \text{if} \quad \mathtt{R}(\mathtt{r}_1) \neq 0 & \text{(E-Bnz)}
\end{array}
$$

Figure 4: Evaluation rules for TAL-0

operand $\mathtt{v}$ into register $\mathtt{r}$, (2) $\mathtt{add}\ \mathtt{r}_1, \mathtt{r}_2, \mathtt{v}$ adds the value of $\mathtt{r}_2$ and $\mathtt{v}$, and puts the result into $\mathtt{r}_1$, and (3) $\mathtt{bz}\ \mathtt{r}\ \mathtt{v}$ branches to operand $\mathtt{v}$ if $\mathtt{r}$ is zero. An operand can be a register $\mathtt{r}$, an integer $\mathtt{n}$, or a program location ℓ. A special instruction $\mathtt{jmp}\ \mathtt{v}$ unconditionally jumps to operand $\mathtt{v}$ and is used to delimit an instruction block. Similar to the stack and register file in Section 2.1, heap $\mathtt{H}$ maps locations to instruction blocks while register file $\mathtt{R}$ maps indices to integers or locations. We write $\mathtt{I} \mid \mathtt{H} \# \mathtt{R}$ for the machine state with instruction block $\mathtt{I}$, heap $\mathtt{H}$ and register file $\mathtt{R}$.

As an example, Figure 3 shows the product function written in TAL-0. The code assumes that the inputs are in registers $\mathtt{r}_1$ and $\mathtt{r}_2$, the output in $\mathtt{r}_3$, and the continuation in $\mathtt{r}_4$. In other words, $\mathtt{r}_4$ holds the location to jump to when the program finishes computing $\mathtt{r}_3 = \mathtt{r}_1 \times \mathtt{r}_2$.

The dynamic semantics of TAL-0 are specified by the evaluation rules of the form $\mathtt{I} \mid \mathtt{H} \# \mathtt{R} \longrightarrow \mathtt{I}' \mid \mathtt{H}' \# \mathtt{R}'$, as shown in Figure 4, which says that machine state $\mathtt{I} \mid \mathtt{H} \# \mathtt{R}$ steps to state $\mathtt{I}' \mid \mathtt{H}' \# \mathtt{R}'$. Note that not all possible machine states have an evaluation rule: for example, there is no rule for $\mathtt{jmp}\ \mathtt{n};\ \mathtt{I} \mid \mathtt{H} \# \mathtt{R}$ (jumping to an integer) or for $\mathtt{add}\ \mathtt{r}_1, \mathtt{r}_2, \ell$ (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 $\mathtt{int}$ as the base type such that the typing rule $\mathtt{H} \# \mathtt{R} \vdash \mathtt{n} : \mathtt{int}$ (T-Int) says that, under any heap type and any register file type, the integer literal $\mathtt{n}$ has the integer type $\mathtt{int}$. Here $\mathtt{H}$ and $\mathtt{R}$ are type contexts that map location ℓ or register $\mathtt{r}$ to type $\mathtt{t}$ (T-Lab and T-Reg). Note that we use the meta-variable $\mathtt{H}$ for both the actual heap of code during evaluation and the heap type context during typing, and similarly $\mathtt{R}$ for register file. When values and types are used in the same rule, we use the convention that $\mathtt{H}_0, \mathtt{R}_0$ refer to values and $\mathtt{H}, \mathtt{R}$ to types (T-Heap, T-Regs and T-State).

For instruction $\mathtt{i}$, we use its precondition on register file type $\mathtt{R}_1$ and its postcondition on register file type $\mathtt{R}_2$ under the heap type $\mathtt{H}$ to assign the type $\mathtt{R}_1 \to \mathtt{R}_2$ (T-Mov, T-Add and

$$
\begin{array}{llcl}
\text{Operand types} & \mathtt{t} & ::= & \mathtt{int} \mid \mathtt{R} \mid \alpha \mid \forall\alpha.\mathtt{t} \\
\text{Heap types} & \mathtt{H} & ::= & \epsilon \mid \mathtt{H}, \ell{:}\mathtt{t} \\
\text{Register file types} & \mathtt{R} & ::= & \epsilon \mid \mathtt{R}, \mathtt{r}{:}\mathtt{t}
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Operand typing} & \mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{v} & : & \mathtt{t} \\
\text{Instruction typing} & \mathtt{H} \vdash \mathtt{i} & : & (\mathtt{R} \rightarrow \mathtt{R}) \\
\text{Block typing} & \mathtt{H} \vdash \mathtt{I} & : & \mathtt{t} \\
\text{Heap typing} & \vdash \mathtt{H}_0 & : & \mathtt{H} \\
\text{Register file typing} & \mathtt{H} \vdash \mathtt{R}_0 & : & \mathtt{R} \\
\text{State typing} & \vdash (\mathtt{I} \mid \mathtt{H}_0 \;\#\; \mathtt{R}_0) & : & \mathtt{H} \;\#\; \mathtt{R}
\end{array}
$$

$$\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{n} : \mathtt{int} \quad \text{(T-Int)}$$

$$\mathtt{H} \;\#\; \mathtt{R} \vdash \ell : \mathtt{H}(\ell) \quad \text{(T-Lab)}$$

$$\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{r} : \mathtt{R}(\mathtt{r}) \quad \text{(T-Reg)}$$

$$\frac{\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{v} : \forall\alpha.\ \mathtt{t}_1}{\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{v} : \mathtt{t}_1[\alpha \mapsto \mathtt{t}_2]} \quad \text{(T-Inst)}$$

$$\frac{\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{v} : \mathtt{t}}{\mathtt{H} \vdash \mathtt{mov}\ \mathtt{r}, \mathtt{v} : (\mathtt{R} \rightarrow \mathtt{R}, \mathtt{r}{:}\mathtt{t})} \quad \text{(T-Mov)}$$

$$\frac{\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{r}_2 : \mathtt{int} \quad \mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{v} : \mathtt{int}}{\mathtt{H} \vdash \mathtt{add}\ \mathtt{r}_1, \mathtt{r}_2, \mathtt{v} : (\mathtt{R} \rightarrow \mathtt{R}, \mathtt{r}_1{:}\mathtt{int})} \quad \text{(T-Add)}$$

$$\frac{\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{r} : \mathtt{int} \quad \mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{v} : \mathtt{R}}{\mathtt{H} \vdash \mathtt{bz}\ \mathtt{r}, \mathtt{v} : (\mathtt{R} \rightarrow \mathtt{R})} \quad \text{(T-Bz)}$$

$$\frac{\mathtt{H} \;\#\; \mathtt{R} \vdash \mathtt{v} : \mathtt{R}}{\mathtt{H} \vdash \mathtt{jmp}\ \mathtt{v} : \mathtt{R}} \quad \text{(T-Jmp)}$$

$$\frac{\mathtt{H} \vdash \mathtt{i} : (\mathtt{R}_1 \rightarrow \mathtt{R}_2) \quad \mathtt{H} \vdash \mathtt{I} : \mathtt{R}_2}{\mathtt{H} \vdash \mathtt{i};\ \mathtt{I} : \mathtt{R}_1} \quad \text{(T-Seq)}$$

$$\frac{\mathtt{H} \vdash \mathtt{I} : \mathtt{t}}{\mathtt{H} \vdash \mathtt{I} : \forall\alpha.\ \mathtt{t}} \quad \text{(T-Gen)}$$

$$\frac{\forall \ell \in \mathtt{dom(H)}.\ \mathtt{H} \vdash \mathtt{H}_0(\ell) : \mathtt{H}(\ell)}{\vdash \mathtt{H}_0 : \mathtt{H}} \quad \text{(T-Heap)}$$

$$\frac{\forall \mathtt{r} \in \mathtt{dom(R)}.\ \mathtt{H} \;\#\; \epsilon \vdash \mathtt{R}_0(\mathtt{r}) : \mathtt{R}(\mathtt{r})}{\mathtt{H} \vdash \mathtt{R}_0 : \mathtt{R}} \quad \text{(T-Regs)}$$

$$\frac{\vdash \mathtt{H}_0 : \mathtt{H} \quad \mathtt{H} \vdash \mathtt{R}_0 : \mathtt{R} \quad \mathtt{H} \vdash \mathtt{I} : \mathtt{R}}{\vdash (\mathtt{I} \mid \mathtt{H}_0 \;\#\; \mathtt{R}_0) : \mathtt{H} \;\#\; \mathtt{R}} \quad \text{(T-State)}$$

Figure 5: Types and typing rules for TAL-0

T-Bz). For instruction block $\mathtt{jmp\ v}$ or $\mathtt{i;\ I}$, we check if the postcondition $\mathtt{R}_2$ of an instruction of type $\mathtt{R}_1 \rightarrow \mathtt{R}_2$ matches the precondition $\mathtt{R}_1$ of the next instruction (T-Jmp and T-Seq). A machine state $\mathtt{I} \mid \mathtt{H}_0;\ \mathtt{R}_0$ is well-typed if its instruction block $\mathtt{I}$, heap $\mathtt{H}_0$, and register file $\mathtt{R}_0$ 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 ℓ_1 can jump to ℓ_2, which may jump back to ℓ_1).

Polymorphic types There remain two types (α and $\forall\alpha.\mathtt{t}$) 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 $\mathtt{jmp\ prod} \mid \mathtt{r}_1 \mapsto 2, \mathtt{r}_2 \mapsto 3, \mathtt{r}_3 \mapsto 0, \mathtt{r}_4 \mapsto \mathtt{halt}\ \#\ \mathtt{H}_0$ where $\mathtt{halt}$ is the location of code to halt the execution and $\mathtt{H}_0$ is the product function in Figure 3. We want to show that the program type-checks under the following heap and register file types:

$$\begin{aligned} \mathtt{H} &= \mathtt{prod{:}R}, \mathtt{loop{:}R}, \mathtt{done{:}R} \\ \mathtt{R} &= \mathtt{r}_1{:}\mathtt{int}, \mathtt{r}_2{:}\mathtt{int}, \mathtt{r}_3{:}\mathtt{int}, \mathtt{r}_4{:}(\forall\alpha.\ \mathtt{r}_1{:}\mathtt{int}, \mathtt{r}_2{:}\mathtt{int}, \mathtt{r}_3{:}\mathtt{int}, \mathtt{r}_4{:}\alpha) \end{aligned}$$

For example, the typing derivation for $\mathtt{H} \vdash \mathtt{bz\ r}_1, \mathtt{done} : (\mathtt{R} \rightarrow \mathtt{R})$ is

$$\dfrac{\dfrac{}{\mathtt{H}\ \#\ \mathtt{R} \vdash \mathtt{r}_1 : \mathtt{int}}\ \text{T-Reg} \qquad \dfrac{}{\mathtt{H}\ \#\ \mathtt{R} \vdash \mathtt{done} : \mathtt{R}}\ \text{T-Lab}}{\mathtt{H} \vdash \mathtt{bz\ r}_1, \mathtt{done} : (\mathtt{R} \rightarrow \mathtt{R})}\ \text{T-Bz}$$

as $\mathtt{R}(\mathtt{r}_1) = \mathtt{int}$ and $\mathtt{H}(\mathtt{done}) = \mathtt{R}$. Derivations for other parts of the program are similar, except that for the last instruction $\mathtt{jmp\ r}_4$. We need to prove that $\mathtt{H}\ \#\ \mathtt{R} \vdash \mathtt{r}_4 : \mathtt{R}$ (T-Jmp). But this requires proving that $\mathtt{R}(\mathtt{r}_4) = \mathtt{R}$, which has no solution in a simple type system. With polymorphic types, we can generalize the type of $\mathtt{r}_4$ to be a type variable α such that $\mathtt{H}\ \#\ \mathtt{R} \vdash \mathtt{r}_4 : (\forall\alpha.\ \mathtt{r}_1{:}\mathtt{int}, \mathtt{r}_2{:}\mathtt{int}, \mathtt{r}_3{:}\mathtt{int}, \mathtt{r}_4{:}\alpha)$ (T-Gen). Only at the instruction $\mathtt{jmp\ r}_4$ is the polymorphic type instantiated to be $(\forall\alpha.\ \mathtt{r}_1{:}\mathtt{int}, \mathtt{r}_2{:}\mathtt{int}, \mathtt{r}_3{:}\mathtt{int}, \mathtt{r}_4{:}\alpha)[\alpha \mapsto \mathtt{R}] = \mathtt{r}_1{:}\mathtt{int}, \mathtt{r}_2{:}\mathtt{int}, \mathtt{r}_3{:}\mathtt{int}, \mathtt{r}_4{:}\mathtt{R}$ (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* $\vdash (\mathtt{I} \mid \mathtt{H}_0\ \#\ \mathtt{R}_0) : \mathtt{H}\ \#\ \mathtt{R}$ *and* $\mathtt{I} \mid \mathtt{H}_0\ \#\ \mathtt{R}_0 \longrightarrow \mathtt{I}' \mid \mathtt{H}_0'\ \#\ \mathtt{R}_0'$*, then* $\vdash (\mathtt{I}' \mid \mathtt{H}_0'\ \#\ \mathtt{R}_0') : \mathtt{H}\ \#\ \mathtt{R}$. *That is, a well-typed program keeps its type during evaluation.*

2. *Progress: If* $\vdash (\mathtt{I} \mid \mathtt{H}_0\ \#\ \mathtt{R}_0) : \mathtt{H}\ \#\ \mathtt{R}$*, then either* $\mathtt{I} = \mathtt{jmp\ halt}$ *or* $\mathtt{I} \mid \mathtt{H}_0\ \#\ \mathtt{R}_0 \longrightarrow \mathtt{I}' \mid \mathtt{H}_0'\ \#\ \mathtt{R}_0'$ *for some* $\mathtt{I}', \mathtt{H}_0', \mathtt{R}_0'$. *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:

$$(\lambda \mathtt{f}\ (\mathtt{x{:}int}).\ \mathtt{ifz}\ \mathtt{x}\ 1\ (\mathtt{x} \times (\mathtt{f}\ (\mathtt{x} - 1))))\ 6$$

We write $\lambda\mathtt{f}\ (\mathtt{x}_1{:}\mathtt{t}_1, \ldots, \mathtt{x}_n{:}\mathtt{t}_n).\ \mathtt{e}$ for a recursive function named $\mathtt{f}$ with typed parameters $\mathtt{x}_1{:}\mathtt{t}_1, \ldots, \mathtt{x}_n{:}\mathtt{t}_n$ and body expression $\mathtt{e}$. The expression $\mathtt{ifz}\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3$ branches to $\mathtt{e}_2$ or $\mathtt{e}_3$ depending whether if $\mathtt{e}_1$ is zero. A function of type $\mathtt{t}_1 \to \ldots \to \mathtt{t}_n \to \mathtt{t}$ takes arguments of type $\mathtt{t}_1, \ldots, \mathtt{t}_n$ and returns a result of type $\mathtt{t}$. The factorial function, for example, has type $\mathtt{int} \to \mathtt{int}$. We write function applications such as $\mathtt{f}\ (\mathtt{x}-1)$ and $(\lambda\mathtt{f}\ldots)\ 6$ 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

```
(λf (x:int, k:int → void).
   ifz x (k 1)
     let x₀ = x − 1 in
     f x₀ (λf₁ (x₁:int). let x₂ = x × x₁ in k x₂))
6 (λf₂ (x₂:int). halt x₂)
```

We introduce a primitive $\mathtt{halt}\ \mathtt{e}$ that stops the program with the result $\mathtt{e}$ (which is $\mathtt{x}_2$ here). The factorial function $\mathtt{f}$ now takes continuation $\mathtt{k}$ as an additional argument and calls $\mathtt{k}\ 1$ for the base case when $\mathtt{x} = 0$. When $\mathtt{x} \neq 0$, we recursively call $\mathtt{f}$ with $\mathtt{x} - 1$ and the continuation $\mathtt{f}_1$ that multiplies the result so far, $\mathtt{x}_1$, with the input $\mathtt{x}$.

We also introduce $\mathtt{let}$-bindings to specify the order of computation. Compared to $\mathtt{f}\ (\mathtt{x} - 1) \ldots$ in System F, the expression $\mathtt{let}\ \mathtt{x}_0 = \mathtt{x} - 1\ \mathtt{in}\ \mathtt{f}\ \mathtt{x}_0 \ldots$ first computes $\mathtt{x} - 1$ and calls $\mathtt{f}$ with this value. We will see how a sequence of $\mathtt{let}$-bindings and function calls can be readily translated into assembly code. Furthermore, we write the type of $\mathtt{f}_2$ to be $\mathtt{int} \to \mathtt{void}$ to emphasize that the function does not return any value. Similarly, $\mathtt{f}$ now has the type $\mathtt{int} \to (\mathtt{int} \to \mathtt{void}) \to \mathtt{void}$.

```
let f = λ(e:<>, x:int, k:t₀).
  ifz x
    (let (α, kk) = unpack k in
      let e₀ = prj₁ kk in
      let k₀ = prj₂ kk in
      let x₀ = 1 in
      k₀ e₀ x₀)
    (let e₀ = <x, k> in
      let kk₀ = <e₀, f₁> in
      let k₀ = pack (<int, t₀>, kk₀):t₀ in
      let x₀ = x − 1 in
      f e₀ x₀ k₀
in
let f₂ = λ(e:<>, x:int).
  halt x
in
let f₁ = λ(e:<int, t₀>, x:int).
  let x₀ = prj₁ e in
  let k₀ = prj₂ e in
  let x₁ = x₀ × x in
  let (α, kk₀) = unpack k₀ in
  let e₁ = prj₁ kk₀ in
  let k₁ = prj₂ kk₀ in
  k₁ e₁ x₁
in
let main = λ().
  let e₀ = <> in
  let kk₀ = <e₀, f₂> in
  let k₀ = pack (<>, kk₀):t₀ in
  let x₀ = 6 in
  f e₀ x₀ k₀
in
main ()
```

Figure 6: Factorial function after closure passing (where $\mathtt{t}_0 = \exists\alpha.\ \alpha \rightarrow \mathtt{int} \rightarrow \mathtt{void}$)

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 $\lambda \mathtt{f}_1\ (\mathtt{x}_1 : \mathtt{int}).\ \mathtt{let}\ \mathtt{x}_2 = \mathtt{x} \times \mathtt{x}_1\ \mathtt{in}\ \mathtt{k}\ \mathtt{x}_2$, 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 $\mathtt{e}_0$ as a tuple of values $\mathtt{e}_0 = \mathtt{<x, k>}$ such that $\mathtt{f}_2$ on the right column will project out $\mathtt{x}_0$ and $\mathtt{k}_0$ as needed. Since all functions are global definitions now, we can define them with top-level `let`-bindings. We also write the computation $\mathtt{f}\ 6\ \mathtt{f}_2$ explicitly as the function `main`.

Separating lexical variables from code, however, complicate the types of continuations. Before closure passing, both $\mathtt{f}_1$ and $\mathtt{f}_2$ have type $\mathtt{int} \rightarrow \mathtt{void}$ and thus the continuation parameter `k` of `f` has type $\mathtt{int} \rightarrow \mathtt{void}$. Now, with the extra environment parameter, $\mathtt{f}_1$ has type $\mathtt{<int, t_0>} \rightarrow \mathtt{int} \rightarrow \mathtt{void}$ while $\mathtt{f}_2$ has type $\mathtt{<>} \rightarrow \mathtt{int} \rightarrow \mathtt{void}$, where $\mathtt{t}_0$ 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 $\mathtt{<int, t_0>}$ into $\mathtt{t}_0$, 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, $\texttt{f}$ packs the type information $\texttt{<int,}\texttt{t}_0\texttt{>}$ with the continuation $\texttt{kk}_0 = \texttt{<e}_0\texttt{,f}_1\texttt{>}$ for the recursive call while $\texttt{main}$ packs $\texttt{<>}$ with $\texttt{kk}_0 = \texttt{<e}_0\texttt{,f}_2\texttt{>}$. To use the continuation, we unpack to find out its type information α in additional to its value $\texttt{kk}$ by the new $\texttt{let}$-binding expression $\texttt{let (}\alpha\texttt{,kk) = unpack k in}$. The continuation parameter $\texttt{k}$ of $\texttt{f}$ can now have the existential type $\texttt{t}_0 = \exists\alpha.\ \alpha \to \texttt{int} \to \texttt{void}$.

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 $\texttt{malloc r:t}$ to allocate heap space large enough for type $\texttt{t}$ and store the pointer into register $\texttt{r}$. Memory safety ensures that locations to heap are always valid. $\texttt{malloc}$ can be implemented by linking to a conservative garbage collector [15]. For example, the instruction block $\texttt{f}_0$ (the first branch of $\texttt{ifz}$) contains the translation of $\texttt{let e}_0 = \texttt{<x,k> in}$: we first allocate space with $\texttt{malloc r}_4\texttt{:<int,t}_1\texttt{>}$, and then initialize fields with $\texttt{store r}_4\texttt{[0],r}_2$ and $\texttt{store r}_4\texttt{[1],r}_3$, assuming that $\texttt{r}_2 = \texttt{x}$ and $\texttt{r}_3 = \texttt{k}$.

Two additional instructions $\texttt{pack r}_1\texttt{,(t}_1\texttt{,r}_2\texttt{):t}_2$ and $\texttt{unpack (}\alpha\texttt{,r}_1\texttt{),r}_2$ are for direct translation of $\texttt{let x}_1 = \texttt{pack (t}_1\texttt{,x}_2\texttt{):t}_2$ and $\texttt{let (}\alpha\texttt{,x}_1\texttt{)} = \texttt{x}_2$. These two instructions simply annotate existential types in type-checking and can be implemented as $\texttt{mov}$.

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 polymorphic 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 continuations 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

$$\forall\alpha_1.\ (\texttt{r}_1\texttt{:}\alpha_1\texttt{,r}_2\texttt{:}(\exists\alpha_2.\ (\alpha_2\texttt{,(r}_1\texttt{:}\alpha_1\texttt{,r}_2\texttt{:}\alpha_2\texttt{)))})$$

```
f:    λ(r1:<>,r2:int,r3:t1).              % let f = λ(e:<>,x:int,k:t0).
      bnz r2,f0                           % ifz x
      unpack (α,r3),r3                    % let (α,kk) = unpack k in
      load r1,r3[0]                       % let e0 = prj1 kk in
      load r4,r3[1]                       % let k0 = prj2 kk in
      mov r2,1                            % let x0 = 1 in
      jmp r4                              % k0 e0 x0

f0:   λ(r1:<>,r2:int,r3:t1).              % let f0 = λ(e:<>,x:int,k:t0).
      malloc r4:<int,t1>                  % let e0 = <x,k> in
      store r4[0],r2                      %
      store r4[1],r3                      %
      malloc r3:t2                        % let kk0 = <e0,f1> in
      store r3[0],r4                      %
      store r3[1],f1                      %
      pack r3,(<int,t1>,r3):t1            % let k0 = pack (<int,t0>,kk0):t0 in
      sub r2,r2,1                         % let x0 = x − 1 in
      jmp f                               % f e0 x0 k0

f1:   λ(r1:<int,t1>,r2:int).              % let f1 = λ(e:<int,t0>,x:int).
      load r3,r1[0]                       % let x0 = prj1 e in
      load r4,r1[1]                       % let k0 = prj2 e in
      mul r2,r3,r2                        % let x1 = x0 × x in
      unpack (α,r4),r4                    % let (α,kk0) = unpack k0 in
      load r1,r4[0]                       % let e1 = prj1 kk0 in
      load r5,r4[1]                       % let k1 = prj2 kk0 in
      jmp r5                              % k1 e1 x1

f2:   λ(r1:<>,r2:int).                    % let f2 = λ(e:<>,x:int).
      mov r1,r2                           % halt x
      halt                                %

main: λ().                                % let main = λ().
      malloc r4:<>                        % let e0 = <> in
      malloc r3:<<>,(r1:<>,r2:int)>       % let kk0 = <e0,f2> in
      store r3[0],r4                      %
      store r3[1],f2                      %
      pack r3,(<>,r3):t1                  % let k0 = pack (<>,kk0):t0 in
      mov r2,6                            % let x0 = 6 in
      jmp f                               % f e0 x0 k0
```

Figure 7: Factorial function in TAL (where $\mathtt{t}_0 = \exists\alpha.\ \alpha \rightarrow \mathtt{int} \rightarrow \mathtt{void}$, $\mathtt{t}_1 = \exists\alpha.\ (\alpha, (\mathtt{r}_1{:}\alpha, \mathtt{r}_2{:}\mathtt{int}))$, and $\mathtt{t}_2 = \mathtt{<<int}, \mathtt{t}_1\mathtt{>}, (\mathtt{r}_1{:}\mathtt{<int}, \mathtt{t}_1\mathtt{>}, \mathtt{r}_2{:}\mathtt{int})\mathtt{>}$)

where α_1 and α_2 are fresh and r_2 contains the continuation, must be parametric in α_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 ($\mathtt{H_0}$), and register files ($\mathtt{R_0}$). We then define instances of operands ($\mathtt{vr}, \mathtt{v_n}, \mathtt{vl}$), instructions (`mov`, `add`, `bz`), blocks (`jmp`, `seq`), heaps (`H0nil`, `H0cons`), and register files (`R0nil`, `R0consn`, `R0consl`). 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 $\mathtt{all} : (\mathtt{t} \to \mathtt{t}) \to \mathtt{t}$.

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

```
r : type.
n : type.
l : type.
v : type.
i : type.
I : type.
H0 : type.
R0 : type.

vr : r -> v.
vn : n -> v.
vl : l -> v.

mov : r -> v -> i.
add : r -> r -> v -> i.
bz : r -> v -> i.
jmp : v -> I.
seq : i -> I -> I.

H0nil : H0.
H0cons : H0 -> l -> I -> H0.
R0nil : R0.
R0consn : R0 -> r -> n -> R0.
R0consl : R0 -> r -> l -> R0.

t : type.
H : type.
R : type.
int : t.
tr : R -> t.
all : (t -> t) -> t.
Hnil : H.
Hcons : H -> l -> t -> H.
Rnil : R.
Rcons : R -> r -> t -> R.

vt : H -> R -> v -> t -> type.
it : H -> i -> R -> R -> type.
It : H -> I -> t -> type.
Ht : H -> H0 -> H -> type.
Rt : H -> R0 -> R -> type.
St : I -> H0 -> R0 -> H -> R -> type.

tint : vt H1 R1 (vn N1) int.
tlab0: vt (Hcons H1 L1 T1) R1 (vl L1) T1.
tlab1: vt (Hcons H1 L1 T1) R1 V1 T2
  <- vt H1 R1 V1 T2.
treg0: vt H1 (Rcons R1 R2 T1) (vr R2) T1.
treg1: vt H1 (Rcons R1 R2 T1) V1 T2
  <- vt H1 R1 V1 T2.

tinst : vt H1 R1 V1 (T1 T2)
  <- vt H1 R1 V1 (all T1).

tmov : it H1 (mov R1 V1) R2 (Rcons R2 R1 T1)
  <- vt H1 R2 V1 T1.
tadd : it H1 (add R1 R2 V1) R3 (Rcons R3 R1 int)
  <- vt H1 R3 (vr R2) int
  <- vt H1 R3 V1 int.
tbz : it H1 (bz R1 V1) R2 R2
  <- vt H1 R2 (vr R1) int
  <- vt H1 R2 V1 (tr R2).

tjmp : It H1 (jmp V1) (tr R1)
  <- vt H1 R1 V1 (tr R1).
tseq : It H1 (seq I1 I2) (tr R1)
  <- it H1 I1 R1 R2
  <- It H1 I2 (tr R2).
tgen : It H1 I1 (all [T1] T2)
  <- It H1 I1 T2.

H0find : H0 -> l -> I -> type.
hf0 : H0find (H0cons H1 L1 I1) L1 I1.
hf1 : H0find (H0cons H1 L1 I1) L2 I2
  <- H0find H1 L2 I2.

theap0 : Ht H1 H2 Hnil.
theap1 : Ht H1 H2 (Hcons H3 L1 T1)
  <- H0find H2 L1 I1
  <- It H1 I1 T1
  <- Ht H1 H2 H3.

R0find : R0 -> r -> v -> type.
rf0n : R0find (R0consn R1 R2 N1) R2 (vn N1).
rf0l : R0find (R0consl R1 R2 L1) R2 (vl L1).
rf1n : R0find (R0consn R1 R2 N1) R3 V1
  <- R0find R1 R3 V1.
rf1l : R0find (R0consl R1 R2 L1) R3 V1
  <- R0find R1 R3 V1.

tregs0 : Rt H1 R1 Rnil.
tregs1 : Rt H1 R1 (Rcons R2 R3 T1)
  <- R0find R1 R3 V1
  <- vt H1 Rnil V1 T1
  <- Rt H1 R1 R2.

tstate : St I1 H1 R1 H2 R2
  <- Ht H2 H1 H2
  <- Rt H2 R1 R2
  <- It H2 I1 (tr R2).
```

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 (`HOfind` and `ROfind` instead of simply `H(ℓ)` or `R(r)`) and to use capital letters for logical variables (such as $\mathtt{H}_1$ and $\mathtt{R}_1$ in rule `tint`).

We can now type-check $\mathtt{H}_1 \mid \mathtt{bz}\ \mathtt{r}_1, \mathtt{done} : \mathtt{R}_1 \rightarrow \mathtt{R}_1$ in Twelf, as we have done manually in Section 3.1, by this additional code:

```
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 (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 $\mathtt{r}_1, \mathtt{r}_2, \mathtt{r}_3, \mathtt{r}_4$ as registers. We need to cons up the list of heap and register file types ($\mathtt{H}_1$ and $\mathtt{R}_1$) 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` $\mathtt{H}_1$ `(bz` $\mathtt{r}_1$ `(vl done))` $\mathtt{R}_1$ $\mathtt{R}_1$ 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 fashion[1]. 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].

[1] `http://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

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *ACM Conference on Programming Language Design and Implementation*, 1998.

[2] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 1985.

[3] Andrew Appel, Edward Felten, and David Walker. Secure internet programming. `http://www.cs.princeton.edu/sip`.

[4] Andrew W. Appel. Foundational Proof-Carrying Code. In *IEEE Symposium on Logic in Computer Science*, 2001.

[5] Andrew W. Appel and Amy P. Felty. A Semantic Model of Types and Machine Instuctions for Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, 2000.

[6] Andrew W. Appel and Trevor Jim. Continuation-Passing, Closure-Passing Style. In *ACM Symposium on Principles of Programming Languages*, 1989.

[7] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* Addison Wesley, 2000.

[8] David Aspinall. Subtyping with Singleton Types. In *Computer Science Logic*, 1994.

[9] Anindya Banerjee and David A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Computer Security Foundations Workshop*, 2002.

[10] Anindya Banerjee and David A. Naumann. Using Access Control for Secure Information Flow in a Java-like Language. In *Computer Security Foundations Workshop*, 2003.

[11] David Basin, Stefan Friedrich, and Marek Gawkowski. Bytecode Verification by Model Checking. *Journal of Automated Reasoning*, 2003.

[12] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java Bytecodes. In *ACM International Conference on Functional Programming*, 1998.

[13] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM Symposium on Operating Systems Principles*, 1995.

[14] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From Region Inference to von Neumann Machines via Region Representation Inference. In *ACM Symposium on Principles of Programming Languages*, 1996.

[15] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software - Practice and Experience*, 1988.

[16] Eduardo Bonelli, Adriana Compagnoni, and Ricardo Medel. SIFTAL: A Typed Assembly Language for Secure Information Flow Analysis, 2004. Unpublished.

[17] Alessandro Cimatti, Fausto Giunchiglia, Paolo Pecchiari, Bruno Pietra, Joe Profeta, Dario Romano, Paolo Traverso, and Bing Yu. A Provably Correct Embedded Verifier for the Certification of Safety Critical Software. In *Computer Aided Verification*, 1997.

[18] Alessandro Coglio. Simple Verification Technique for Complex Java Bytecode Subroutines. In *ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002.

[19] Christopher Colby, Peter Lee, and George C. Necula. A Proof-Carrying Code Architecture for Java. In *Computer Aided Verification*, 2000.

[20] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *ACM Conference on Programming Language Design and Implementation*, 2000.

[21] Karl Crary. Foundations for the Implementation of Higher-Order Subtyping. In *ACM International Conference on Functional Programming*, 1997.

[22] Karl Crary. Toward a foundational typed assembly language. In *ACM Symposium on Principles of Programming Languages*, 2003.

[23] Karl Crary and Greg Morrisett. Typed Assembly Language Compiler. `http://www.cs.cornell.edu/talc`.

[24] Karl Crary, David Walker, and J. Gregory Morrisett. Typed Memory Management in a Calculus of Capabilities. In *ACM Symposium on Principles of Programming Languages*, 1999.

[25] Karl Crary and Stephanie Weirich. Resource Bound Certification. In *ACM Symposium on Principles of Programming Languages*, 2000.

[26] Olivier Danvy and Andrzej Filinski. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 1992.

[27] Robert DeLine and Manuel Fahndrich. Enforcing High-Level Protocols in Low-Level Software. In *ACM Conference on Programming Language Design and Implementation*, 2001.

[28] Gilles Dowek and Christine Paulin-Mohring. The Coq Project. `http://coq.inria.fr`.

[29] Manuel Fahndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *ACM Conference on Programming Language Design and Implementation*, 2002.

[30] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 1967.

[31] Cedric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. In *ACM Symposium on Principles of Programming Languages*, 2002.

[32] Stephen N. Freund and John C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *ACM Conference on Object Oriented Programming Systems Languages and Applications*, 1999.

[33] Stephen N. Freund and John C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 2003.

[34] Jean-Yves Girard. *Interpretation Functionelle et Elimination des Coupures dans l'Arithmetique d'Order Superieure*. PhD thesis, Universite Paris VII, 1972.

[35] Allen Goldberg. A Specification of Java Loading and Bytecode Verification. In *ACM Conference on Computer and Communications Security*, 1998.

[36] Dan Grossman and J. Gregory Morrisett. Scalable Certification for Typed Assembly Language. In *Types in Compilation*, 2000.

[37] Masami Hagiya and Akihiko Tozawa. On a New Method for Dataflow Analysis of Java Virtual Machine Subroutines. In *Static Analysis Symposium*, 1998.

[38] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A Syntactic Approach to Foundational Proof-Carrying Code. In *IEEE Symposium on Logic in Computer Science*, 2002.

[39] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A Framework for Defining Logics. In *IEEE Symposium on Logic in Computer Science*, 1987.

[40] Robert Harper, Peter Lee, and Frank Pfenning. The Fox Project. `http://www.cs.cmu.edu/~fox`.

[41] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *ACM Symposium on Principles of Programming Languages*, 2002.

[42] Thomas P. Jensen, Daniel Le Metayer, and Tommy Thorn. Verification of Control Flow based Security Properties. In *IEEE Symposium on Security and Privacy*, 1999.

[43] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Usenix Annual Technical Conference*, 2002.

[44] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. `http://www.haskell.org`.

[45] Assaf Kfoury and Joe Wells. The Church Project. `http://types.bu.edu`.

[46] G. Klein and M. Wildmoser. Verified Bytecode Subroutines. *Journal of Automated Reasoning*, 2003.

[47] Xavier Leroy. The Ocaml Programming Language. `http://caml.inria.fr`.

[48] Xavier Leroy. Unboxed Objects and Polymorphic Typing. In *ACM Symposium on Principles of Programming Languages*, 1992.

[49] Xavier Leroy. Bytecode verification on Java smart cards. *Software - Practice and Experience*, 2002.

[50] Xavier Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 2003.

[51] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1999.

[52] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*, 1993.

[53] John L. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, 1962.

[54] Gary Meehan and Mike Joy. Compiling Lazy Functional Programs to Java Bytecode. *Software - Practice and Experience*, 1999.

[55] Microsoft. *ECMA-334: C♯ Language Specification*. European Computer Manufacturers Association, 2002.

[56] Microsoft. *ECMA-335: Common Language Infrastructure*. European Computer Manufacturers Association, 2002.

[57] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. `http://www.smlnj.org`.

[58] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed Closure Conversion. In *ACM Symposium on Principles of Programming Languages*, 1996.

[59] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[60] Greg Morrisett. *Advanced Topics in Types and Programming Languages*, chapter Typed Assembly Language. MIT Press, 2004.

[61] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A Realistic Typed Assembly Language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, 1999.

[62] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *ACM Symposium on Principles of Programming Languages*, 1998.

[63] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[64] George Necula. *Advanced Topics in Types and Programming Languages*, chapter Proof-Carrying Code. MIT Press, 2004.

[65] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1997.

[66] George C. Necula. Proof-Carrying Code. In *ACM Symposium on Principles of Programming Languages*, 1997.

[67] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Operating Systems Design and Implementation*, 1996.

[68] George C. Necula and Peter Lee. Efficient Representation and Validation of Proofs. In *IEEE Symposium on Logic in Computer Science*, 1998.

[69] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. In *ACM Conference on Programming Language Design and Implementation*, 1998.

[70] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, 2002.

[71] George C. Necula and Shree Prakash Rahul. Oracle-based checking of untrusted software. In *ACM Symposium on Principles of Programming Languages*, 2001.

[72] Tobias Nipkow. Java Bytecode Verification. *Journal of Automated Reasoning*, 2003.

[73] Robert O'Callahn. A Simple, Comprehensive Type System for Java Bytecode Subroutines. In *ACM Symposium on Principles of Programming Languages*, 1999.

[74] Larry Paulson and Tobias Nipkow. The Isabelle Project. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[75] Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In *ACM Symposium on Principles of Programming Languages*, 2003.

[76] Frank Pfenning and Conal Elliot. Higher-Order Abstract Syntax. In *ACM Conference on Programming Language Design and Implementation*, 1988.

[77] Frank Pfenning and Carsten Schurmann. The Twelf Project. `http://www.twelf.org`.

[78] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[79] Benjamin C. Pierce. Types and Programming Languages: The Next Generation. In *IEEE Symposium on Logic in Computer Science*, 2003.

[80] Francois Pottier, Christian Skalka, and Scott F. Smith. A Systematic Approach to Static Access Control. In *European Symposium on Programming*, 2001.

[81] John C. Reynolds. Towards a Theory of Type Structure. In *Symposium on Programming*, 1974.

[82] Ekkehard Rohwedder and Frank Pfenning. Mode and Termination Checking for Higher-Order Logic Programs. In *European Symposium on Programming*, 1996.

[83] Eva Rose and Kristoffer Rose. Lightweight Bytecode Verification. In *OOPSLA Workshop on Formal Underpinnings of Java*, 1998.

[84] David A. Schmidt. Data Flow Analysis is Model Checking of Abstract Interpretations. In *ACM Symposium on Principles of Programming Languages*, 1998.

[85] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 2000.

[86] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A Language-Based Approach to Security. In *Informatics*, 2001.

[87] Bernard P. Serpette and Manuel Serrano. Compiling scheme to JVM bytecode: : a performance study. In *ACM International Conference on Functional Programming*, 2002.

[88] Zhong Shao. The Flint Project. `http://flint.cs.yale.edu`.

[89] Zhong Shao and Andrew W. Appel. A Type-Based Compiler for Standard ML. In *ACM Conference on Programming Language Design and Implementation*, 1995.

[90] Robert F. Stark and Joachim Schmid. Completeness of a Bytecode Verifier and a Certifying Java-to-JVM Compiler. *Journal of Automated Reasoning*, 2003.

[91] Raymie Stata and Martin Abadi. A Type System for Java Bytecode Subroutines. In *ACM Symposium on Principles of Programming Languages*, 1998.

[92] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. In *ACM Conference on Object Oriented Programming Systems Languages and Applications*, 2001.

[93] David Tarditi, J. Gregory Morrisett, P. Cheng, C. Stone, Robert Harper, and Peter Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, 1996.

[94] Joseph Vanderwaart and Karl Crary. A typed interface for garbage collection. In *ACM Types In Languages Design And Implementation*, 2003.

[95] Joseph C. Vanderwaart and Karl Crary. Foundational Typed Assembly Language for Grid Computing. Technical Report CMU-CS-04-104, Carnegie Mellon University, 2004.

[96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 1996.

[97] Philip Wadler. Theorems for Free! In *Functional Programming Languages and Computer Architecture*, 1989.

[98] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *ACM Symposium on Operating Systems Principles*, 1993.

[99] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, 1998.

[100] Hongwei Xi. Imperative Programming with Dependent Types. In *IEEE Symposium on Logic in Computer Science*, 2000.

[101] Hongwei Xi and Robert Harper. A Dependently Typed Assembly Language. In *ACM International Conference on Functional Programming*, 2001.

[102] Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *ACM Conference on Programming Language Design and Implementation*, 1998.