Analysis of Low-Level Code Using Cooperating Decompilers

Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula

University of California, Berkeley, California, USA
{bec,matth,necula}@cs.berkeley.edu

Abstract. We present a modular framework for building assembly-language program analyzers by using a pipeline of decompilers that gradually lift the level of the language to something appropriate for source-level analysis tools. Each decompilation stage contains an abstract interpreter that encapsulates its findings about the program by translating the program into a higher-level intermediate language. For the hardest decompilation tasks a decompiler may request information from higher-level stages in the pipeline. We provide evidence for the modularity of this framework through the implementation of multiple decompilation pipelines for both x86 and MIPS assembly produced by `gcc`, `gcj`, and `coolc` (a compiler for a pedagogical mini-Java language) that share several low-level components. Finally, we discuss our experimental results that apply the BLAST model checker for C and the Cqual analyzer to decompiled assembly.

1 Introduction

There is a growing interest in applying software-quality tools to low-level representations of programs, such as intermediate or virtual-machine languages, or even on native machine code. We want to be able to analyze code whose source is either not available (e.g., libraries) or not easily analyzable (e.g., programs written in languages with complex semantics such as C++, or programs that contain inline assembly). This allows us to analyze the code that is actually executed to ignore possible compilation errors or arbitrary interpretations of underspecified source-language semantics. Many source-level analyses have been ported to low-level code, including type checkers [MWCG99,LY97,CCNS05], program analyzers [Riv03,BR04], model checkers [BRK+05], and program verifiers [CLN+00,BL05]. In our experience, these tools mix the reasoning about high-level notions with the logic for understanding low-level implementation details that are introduced during compilation, such as stack frames, calling conventions, exception implementation, and data layout. We would like to segregate the low-level logic into separate modules, to allow for easier sharing between tools and for a cleaner interface with the client analyses. To better understand this issue, consider developing a type checker similar to the Java bytecode verifier but for assembly language. Such a tool has to reason not only about the Java type system, but also the layout of objects, calling conventions, stack frames, with

all the low-level invariants that the compiler intends to preserve. We reported earlier [CCNS05] on such a tool where all of this reasoning is done simultaneously by one module. But such situations arise not just for type checking but essentially for all analyses on assembly language.

In this paper we propose an architecture that modularizes the reasoning about low-level details into separate components. Such a separation of low-level logic has previously been done to a certain degree in tools such as CodeSurfer/x86 [BR04] and Soot [VRCG+99], which expose to client analyses an API for obtaining information about the low-level aspects of the program. In this paper, we adopt a more radical approach in which the low-level logic is packaged as a *decompiler* whose output is an intermediate language that abstracts the low-level implementation details introduced by the compiler. In essence, we propose that an easy way to reuse source-level analysis tools for low-level code is to decompile the low-level code to a level appropriate for the tool. We make the following contributions:

- We propose a decompilation architecture as a way to apply source-level tools to assembly language programs (Sec. 2). The novel aspect of our proposal is that we use decompilation not only to separate the low-level logic from the source-level client analysis, but also as a way to modularize the low-level logic itself. Decompilation is performed by a series of decompilers connected by intermediate languages. We provide a *cooperation* mechanism in order to deal with certain complexities of decompilation.
- We provide evidence for the modularity of this framework through the implementation of multiple decompilation pipelines for both x86 and MIPS assembly produced by `gcc` (for C), `gcj` (for Java), and `coolc` (for Cool [Aik96], a Java-like language used for teaching) that share several low-level components (Sec. 3). We compare the size of the modules with a monolithic assembly-level analysis.
- We demonstrate it is possible to apply source-level tools to assembly code using decompilation by applying the BLAST model checker [HJM+02] and the Cqual analyzer [FTA02] with our `gcc` decompilation pipeline (Sec. 4).

Note that while ideally we would like to apply analysis tools to machine code binaries, we leave the difficult issue of lifting binaries to assembly to other work (perhaps by using existing tools like IDAPro [IDA] as in CodeSurfer/x86 [BR04]).

Challenges. Just like in a compiler, a pipeline architecture improves modularity of the code and allows for easy reuse of modules for different client-analyses. Fig. 1 shows an example of using decompilation modules to process code that has been compiled with the GNU Compiler Collection compilers. Each stage recovers an abstraction that a corresponding compilation stage has concretized. For example, we have a decompiler that decompile the notion of the run-time stack of activation records into the abstraction of functions with local variables (Locals). We use an object-oriented module (OO) to decompile generic object-oriented features (e.g., virtual method dispatch). Finally, the additional type inference

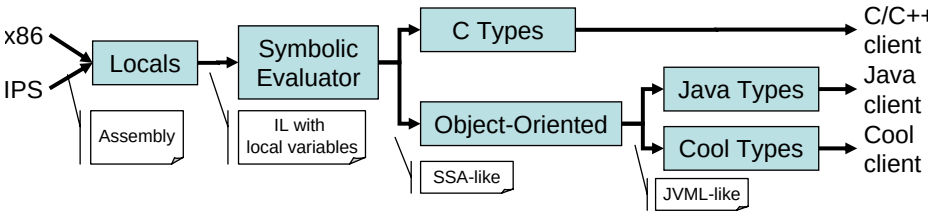

Fig. 1. Cooperating decompilers for the output of `gcc` and `gcj`. Shaded boxes are decompiler modules that produce successively more abstract versions of the program.

modules are able to produce valid source-level programs (except for eliminating `goto`s in the case of Java). We also show that, just like during compilation, global value numbering and static-single assignment greatly facilitate analysis of low-level code and package these transformations as a decompiler (SymEval).

The analogy with compilers is very useful but not sufficient. Compilation is in many respects a many-to-one mapping and thus not easily invertible. Many source-level variables are mapped to the same register, many source-level concepts are mapped to the run-time stack, many source-level operations are mapped to a particular low-level instruction kind. We address this issue by providing each decompiler with additional information about the instruction being decompiled. Some information is computed by the decompiler itself using data-flow analysis. For example, the Locals decompiler can keep track of the value of the stack and frame pointer registers relative to function entry.

The real difficulty is that some information must be provided by higher-level modules. For example, the Locals module must identify all calls and determine the number of arguments, but only the object-oriented module (OO) should understand virtual method invocation. There is a serious circularity here. A decompiler needs information from higher-level decompilers to produce the input for the higher-level decompiler. We introduce a couple of mechanisms to address this problem. First, the entire pipeline of decompilers is executed one instruction at a time. That is, we produce decompiled programs simultaneously at all levels. This setup gives each decompiler the opportunity to accumulate data-flow facts that are necessary for decompiling the subsequent instructions and allows the control-flow graph to be refined as the analysis proceeds. When faced with an instruction that can be decompiled in a variety of ways, a decompiler can consult its own data-flow facts and can also query higher-level decompilers for hints based on their accumulated data-flow facts. Thus it is better to think of decompilers not as stages in a pipeline but as cooperating decompilers. The net result is essentially a reduced product analysis [CC79] on assembly; we explain the benefits of this framework compared to prior approaches based on our previous experiences in Sec. 3 and 5.

```
static int length(List x) {
  int len = 0;
  while (x.hasNext()) {
    x = x.next();
    len++;
  }
  return len;
}
```

Fig. 2. A simple Java method.

2 Cooperating Decompilation Framework

For concreteness, we describe the methodology through an example series of decompiler modules that together are able to perform Java type checking on assembly language. We focus here on the Java pipeline (rather than C), as the desired decompilation is higher-level and thus more challenging to obtain. Consider the example Java program in Fig. 2 and the corresponding assembly code shown in Fig. 3(a). For clarity, we often use register names that are indicative of the source variables to which they correspond (e.g., $\mathbf{r}_x$) or the function they serve (e.g., $\mathbf{r}_{sp}$ for the stack pointer). In this figure, we use the stack and calling conventions from the x86 architecture where the stack pointer $\mathbf{r}_{sp}$ points to the last used word, parameters are passed on the stack, return values are passed in $\mathbf{r}_1$, and $\mathbf{r}_2$ is a callee-save register. self pointer (`this` in Java) is passed as the first argument. Typically, a virtual method dispatch is translated to several lines of assembly: a null-check on the receiver object, looking up the dispatch table and then the method in the dispatch table, passing the receiver object and any other arguments, and finally an indirect jump-and-link (`icall`). To ensure that the `icall` is a correct compilation of a virtual method dispatch, dependencies between assembly instructions must be carefully tracked, such as the requirement that the argument passed as the self pointer is the same (or at least has the same dynamic type) as the object from which the dispatch table is obtained (cf., [LST02,CCNS05]). These difficulties are only exacerbated with instruction reordering and other optimizations. For example, consider the assembly code for the method dispatch to `x.next()` (lines 15–17). Variable `x` is kept in a stack slot ($\mathbf{m}[\mathbf{r}_{sp} + 16]$ at line 15). A small bit of optimization has eliminated the null-check and the re-fetching of the dispatch table of `x`, as a null-check was done on line 6 and the dispatch table was kept in a callee-save register $\mathbf{r}_2$, so clearly some analysis is necessary to decompile it into a method call.

The rest of Fig. 3 shows how this assembly code is decompiled by our system. In summary, the Locals module decompiles stack and calling conventions to provide the abstraction of functions with local variables. The SymEval decompiler performs symbolic evaluation to accumulate and normalize larger expressions to present the program in a source-like SSA form. Object-oriented features, like virtual method dispatch, are identified by the OO module, which must understand implementation details like object layout and dispatch tables. Finally, JavaTypes

	(a) Assembly	(b) Locals	(c) SymEval	(d) OO	(e) JavaTypes
1	length:	length($\mathbf{t}_x$):	length(α_x):	length(α_x : obj):	length(α_x : *List*):
2	...				
3	$\mathbf{m}[\mathbf{r}_{sp}] := 0$	$\mathbf{t}_{len} := 0$	$\alpha_{len} = 0$	$\alpha_{len} = 0$	$\alpha_{len} = 0$
4	$\mathrm{L}_{\mathrm{loop}}$:	$\mathrm{L}_{\mathrm{loop}}$:	$\mathrm{L}_{\mathrm{loop}}$: $\alpha''_{len} = \phi(\alpha_{len}, \alpha'_{len})$ $\alpha''_x = \phi(\alpha_x, \alpha'_x)$	$\mathrm{L}_{\mathrm{loop}}$: $\alpha''_{len} = \phi(\alpha_{len}, \alpha'_{len})$ $\alpha''_x = \phi(\alpha_x, \alpha'_x)$	$\mathrm{L}_{\mathrm{loop}}$: $\alpha''_{len} = \phi(\alpha_{len}, \alpha'_{len})$ $\alpha''_x = \phi(\alpha_x, \alpha'_x)$
5	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_{sp} + 12]$	$\mathbf{r}_1 := \mathbf{t}_x$			
6	`jzero` $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{exc}}$	`jzero` $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{exc}}$	`if` (α''_x=0) $\mathrm{L}_{\mathrm{exc}}$	`if` (α''_x=0) $\mathrm{L}_{\mathrm{exc}}$	`if` (α''_x=0) $\mathrm{L}_{\mathrm{exc}}$
7	$\mathbf{r}_2 := \mathbf{m}[\mathbf{r}_1]$	$\mathbf{r}_2 := \mathbf{m}[\mathbf{r}_1]$			
8	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2 + 32]$	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2 + 32]$			
9	$\mathbf{r}_{sp} := \mathbf{r}_{sp} - 4$				
10	$\mathbf{m}[\mathbf{r}_{sp}] := \mathbf{m}[\mathbf{r}_{sp} + 16]$	$\mathbf{t}_1 := \mathbf{t}_x$			
11	`icall` [$\mathbf{r}_1$]	$\mathbf{r}_1 :=$ `icall` [$\mathbf{r}_1$]($\mathbf{t}_1$)	$\alpha_{rv_1} =$ `icall` [$\mathbf{m}[\mathbf{m}[\alpha''_x] + 32]$] ($\alpha''_x$)	$\alpha_{rv_1} =$ `invokevirtual` [α''_x, 32]()	$\alpha_{rv_1} =$ α''_x.`hasNext`()
12	$\mathbf{r}_{sp} := \mathbf{r}_{sp} + 4$				
13	`jzero` $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{end}}$	`jzero` $\mathbf{r}_1$, $\mathrm{L}_{\mathrm{end}}$	`if` (α_{rv_1}=0) $\mathrm{L}_{\mathrm{end}}$	`if` (α_{rv_1}=0) $\mathrm{L}_{\mathrm{end}}$	`if` (α_{rv_1}=0) $\mathrm{L}_{\mathrm{end}}$
14	$\mathbf{r}_{sp} := \mathbf{r}_{sp} - 4$				
15	$\mathbf{m}[\mathbf{r}_{sp}] := \mathbf{m}[\mathbf{r}_{sp} + 16]$	$\mathbf{t}_1 := \mathbf{t}_x$			
16	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2 + 28]$	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_2 + 28]$			
17	`icall` [$\mathbf{r}_1$]	$\mathbf{r}_1 :=$ `icall` [$\mathbf{r}_1$]($\mathbf{t}_1$)	$\alpha_{rv_2} =$ `icall` [$\mathbf{m}[\mathbf{m}[\alpha''_x] + 28]$] ($\alpha''_x$)	$\alpha_{rv_2} =$ `invokevirtual` [α''_x, 28]()	$\alpha_{rv_2} =$ α''_x.`next`()
18	$\mathbf{r}_{sp} := \mathbf{r}_{sp} + 4$				
19	$\mathbf{m}[\mathbf{r}_{sp} + 12] := \mathbf{r}_1$	$\mathbf{t}_x := \mathbf{r}_1$	$\alpha'_x = \alpha_{rv_2}$	$\alpha'_x = \alpha_{rv_2}$	$\alpha'_x = \alpha_{rv_2}$
20	`incr` $\mathbf{m}[\mathbf{r}_{sp}]$	`incr` $\mathbf{t}_{len}$	$\alpha'_{len} = \alpha''_{len} + 1$	$\alpha'_{len} = \alpha''_{len} + 1$	$\alpha'_{len} = \alpha''_{len} + 1$
21	`jump` $\mathrm{L}_{\mathrm{loop}}$	`jump` $\mathrm{L}_{\mathrm{loop}}$	`jump` $\mathrm{L}_{\mathrm{loop}}$	`jump` $\mathrm{L}_{\mathrm{loop}}$	`jump` $\mathrm{L}_{\mathrm{loop}}$
22	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:	$\mathrm{L}_{\mathrm{end}}$:
23	$\mathbf{r}_1 := \mathbf{m}[\mathbf{r}_{sp}]$	$\mathbf{r}_1 := \mathbf{t}_{len}$			
24	...				
25	`return`	`return` $\mathbf{r}_1$	`return` α''_{len}	`return` α''_{len}	`return` α''_{len}

Fig. 3. Assembly code for the program in Fig. 2 and the output of successive decompilers. The function's prologue and epilogue have been elided. Jumping to $\mathrm{L}_{\mathrm{exc}}$ will trigger a Java `NullPointerException`.

can do a straightforward type analysis (because its input is so high-level) to recover essentially Java with unstructured control-flow.

As can be seen in Fig. 3, one key element of analyzing assembly code is decoding the run-time stack. An assembly analyzer must be able to identify function calls and returns, recognize memory operations as either stack accesses or heap accesses, and must ensure that stack-overflow and calling conventions are handled appropriately. This handling ought to be done in a separate module both because it is not specific to the desired analysis and also to avoid such low-level concerns when thinking about the analysis algorithm (e.g., Java type-checking). In our example decompiler pipeline (Fig. 1), the Locals decompilers handle all of these low-level aspects. On line 17, the Locals decompiler determines that this instruction is a function call with one argument (for now, we elide the details how this done, see the Bidirectional Communication subsection and Fig. 4) and interprets the calling convention to output a function call with one argument that places its return value in $\mathbf{r}_{rv}$. Also, observe that Locals decompiles reads of

and writes to stack slots that are used as local variables into uses of *temporaries* (e.g., $\mathbf{t}_x$) (lines 3, 5, 10, 15, 19, 20, 23). To do these decompilations, the Locals decompiler needs to perform analysis to track, for example, pointers into the stack (we write $\mathsf{sp}(n)$ for a stack pointer that is equal to $\mathbf{r}_{sp}$ on function entry plus n). For instance, Locals needs this information to identify the reads on both lines 5 and 10 as reading the same stack slot $\mathbf{t}_x$. Sec. 3 gives more details about how these decompilers are implemented.

Decompiler Interface. Program analyses are almost always necessary to establish the prerequisites for sound decompilations. We build on the traditional notions of data-flow analysis and abstract interpretation [CC77], which provides a systematic framework for the construction of program analyses. Standard ways to combine abstract interpreters typically rely on all interpreters working on the same language. Instead, we propose here an approach in which the communication mechanism consists of successive decompilations. A lower-level analysis communicates the essence of the information it has discovered as part of the translated instructions to be analyzed by higher-level analyses.

In the remainder of this section, we present signatures for decompilers and intermediate languages. To motivate our design, we evolve them slowly beginning from traditional notions of abstract interpretation. A decompiler operates on instructions from input language and produces instructions in the output language. A language definition implements the following (partial) signature, which includes a type of instructions:

type `instr` . LANGUAGE

We specify for each type or value declaration whether they belong to the LANGUAGE or DECOMPILER signatures. The type of the flow function (i.e., abstract transition relation) a decompiler must implement is as follows:

val `step` : `curr` $\times$ $\texttt{instr}_{in} \to \texttt{instr}_{out} \times$ `succ list` DECOMPILER

for some input language $\texttt{instr}_{in}$ and some output language $\texttt{instr}_{out}$. The type `curr` represents the current abstract state at the given instruction, and `succ` represents a pair of a program location (`loc`) and the abstract successor state for that location, that is,

type `abs`
type `curr` = `abs`
type `succ` = `loc` $\times$ `abs` . DECOMPILER

For our purposes, a program location is either the "fall-through" location or a particular label ℓ. For simplicity in presentation, we assume a decompiler translates one input instruction to one output instruction. Our implementation extends this to allow one-to-many or many-to-one translations.

As part of the framework, we provide a standard top-level fixpoint engine that ensures the exploration of all reachable instructions. To implement this

fixpoint engine, we require in the signature the standard partial ordering and widening operators [CC77]:

```
val ⊑ : abs × abs → bool
val ▽ : abs × abs → abs .                                    DECOMPILER
```

The widening operator yields an abstract domain element that is an upper bound of its inputs such that for any ascending chain $B_0 \sqsubseteq B_1 \sqsubseteq \cdots$, the ascending chain

$$A \sqsubseteq (A \triangledown B_0) \sqsubseteq ((A \triangledown B_0) \triangledown B_1) \sqsubseteq \cdots$$

stabilizes after a finite number of steps.

For simple examples where the necessary communication is unidirectional (that is, from lower-level decompilers to higher-level decompilers via the decompiled instructions), an exceedingly simple composition strategy suffices where we run each decompiler completely to fixpoint gathering the entire decompiled program before running the next one (i.e., a strict pipeline architecture). This architecture does not require a product abstract domain and would be more efficient than one. Unfortunately, as we have alluded to earlier, unidirectional communication is insufficient: lower-level decompilers depend on the analyses of higher-level decompilers to perform their decompilations. We give examples of such situations and describe how to resolve this issue in the following subsection.

Bidirectional Communication. In this subsection, we motivate two complimentary mechanisms for communicating information from higher-level decompilers to lower-level ones. In theory, either mechanism is sufficient for all high-to-low communication but at the cost of efficiency or naturalness. As soon as we consider high-to-low communication, clearly the strict pipeline architecture described above is insufficient: higher-level decompilers must start before lower-level decompilers complete. To address this issue, we run the entire pipeline of decompilers one instruction at a time, which allows higher-level decompilers to analyze the preceding instructions before lower-level decompilers produce subsequent instructions. For this purpose, we provide a product decompiler whose abstract state is the product of the abstract states of the decompilers, but in order to generate its successors, it must string together calls to `step` on the decompilers in the appropriate order and then collect together the abstract states of the decompilers.

Queries. Consider again the dynamic dispatch on line 17 of Fig. 3. In order for the Locals module to (soundly) abstract stack and calling conventions into functions with local variables, it must enforce basic invariants, such as a function can only modify stack slots (used as temporaries) in its own activation record (i.e., stack frame). To determine the extent of the callee's activation record, the Locals module needs to know, among other things, the number arguments of the called function, but only the higher-level decompiler that knows about the class hierarchy (JavaTypes) can determine the calling convention of the methods that

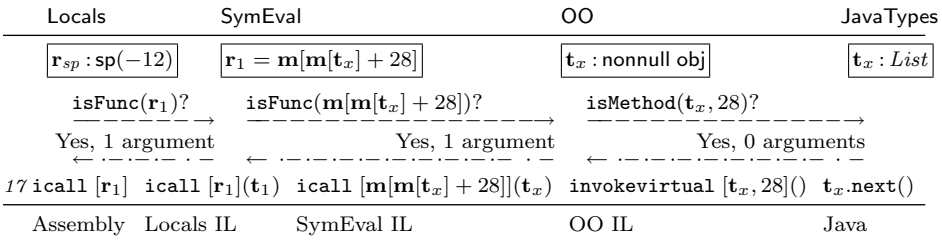

Fig. 4. Queries to resolve the dynamic dispatch from line 17 of Fig. 3. Relevant portions of the abstract states before the `icall` are shown boxed.

$\mathbf{r}_1$ can possibly point to. As we have alluded to earlier, we resolve this issue by allowing lower-level decompilers to query higher-level decompilers for hints. In this case, Locals asks: "Should `icall` [$\mathbf{r}_1$] be treated as a standard function call; if so, how many arguments does it take?". If some higher-level decompiler knows the answer, then it can translate the `icall` to a higher-level call with arguments and a return register and appropriately take into account its possible effects.

In Fig. 4, we show this query process in further detail. We show the decompilers for Locals, symbolic evaluation (SymEval), object-oriented features (OO), and Java features (JavaTypes), eliding the return value. Precisely how these decompilers work is not particularly relevant here (see details in Sec. 3). Focus on the original query `isFunc`($\mathbf{r}_1$) from Locals. To obtain an answer, the query gets decompiled into appropriate variants on the way up to JavaTypes. The answer is then translated on the way down. For the OO module the method has no arguments, but at the lower-level the implicit `this` argument becomes explicit. For JavaTypes to answer the query, it must know the type of the receiver object, which it gets from its abstract state (shown above the queries). The abstract states of the intermediate decompilers are necessary in order to translate queries so that JavaTypes can answer them. Such a query (along with tracking of return addresses) also allows Locals to decompile calls that are implemented in assembly as (indirect) `jump`s (e.g., tail calls), which then allows higher-level decompilers to treat calls uniformly.

To implement queries as shown, we include in the signature LANGUAGE a type of callback objects (`hints`) whose implementations are functions provided by higher-level decompilers. For example, the Calls IL would include a callback for identifying function calls:

$$\begin{array}{l}\textbf{type}\ \texttt{expr} \ = \ \ldots \\ \textbf{type}\ \texttt{hints} = \{\ \texttt{isFunc}\colon \texttt{expr} \to \texttt{bool}\ \}\end{array} \qquad \boxed{\mathsf{CallsIL} : \textsc{Language}}$$

where `expr` is the type of machine expressions for Calls IL.

Intuitively, an object of type `hints` in the output language of a decompiler provides information about the current abstract states of higher-level decompilers. Such an object is provided as an input to the `step` function of each

decompiler; we do this by modifying the `curr` type:

$$\textbf{type}\ \texttt{curr} \quad = \texttt{hints}_{out} \times \texttt{abs}$$
$$\textbf{val}\ \texttt{getHints} : \texttt{hints}_{out} \times \texttt{abs} \to \texttt{hints}_{in}\ . \qquad \boxed{\textsc{Decompiler}}$$

Additionally, each decompiler provides a function `getHints` to create a callback object for lower-level decompilers, based on its current abstract state and on the callback object for the higher-level decompilers. The resulting callback object may operate in one of two ways. When posed a question by the lower-level decompiler, it may obtain the necessary response by examining its current abstract state. For example, in Fig. 4, the JavaTypes decompiler answers the `isFunc` question directly. The alternative is to decompile the question into a question that can be posed to the higher-level decompiler by means of the $\texttt{hints}_{out}$ object. The response might have to be translated back to match the responses expected by the $\texttt{hints}_{in}$ object. The translation of both the questions and the responses can be done using the current abstract state, as shown for the Locals, SymEval, and OO decompilers in Fig. 4.

This architecture with decompilations and callbacks works quite nicely, as long as the decompilers agree on the number of successors and their program locations. In this situation, the job of the product domain is straightforward. In some cases, however, it is convenient to use decompilers that do not always agree on the successors.

Decompiling Control-Flow. Obtaining a reasonable control-flow graph on which to perform analysis is a well-known problem when dealing with assembly code and is often a source of unsoundness, particularly when handling indirect control-flow. For example, switch tables, function calls, function returns, exception raises may all be implemented as indirect jumps (`ijump`) in assembly. We approach this problem by integrating the control-flow determination with the decompilation; that is, we make no *a priori* guesses on where an indirect jump goes and rely on the decompiler modules to resolve them to a set of concrete program points. In general, there are two cases where the decompilers may not be able to agree on the same successors:

1. *Don't Know the Successors.* Sometimes a low-level decompiler does not know the possible concrete successors. For example, if the Locals decompiler cannot resolve an indirect jump, it will produce an *indirect* successor indicating it does not know where the indirect jump will go. However, a higher-level decompiler may be able to refine the indirect successor to a set of concrete successors (that, for soundness, must cover where the indirect jump may actually go). It is then an error if any indirect successors remain unresolved after the entire pipeline.
2. *Additional Successors.* A decompiler may also need to introduce additional successors not known to lower-level modules. For example, an exceptions decompiler may need to express that a function call has not only the normal successor at the following instruction, but also an exceptional successor at the enclosing exception handler.

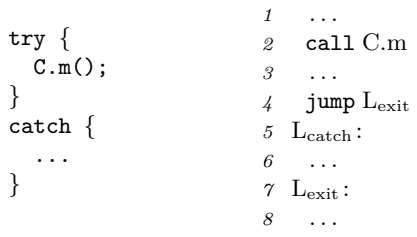

Fig. 5. Compilation of exception handling.

In both examples, a high-level decompiler augments the set of successors with respect to those of the low-level decompilers. The problem is that we do not have abstract states for the low-level decompilers at the newly introduced successors. This, in turn, means that it will be impossible to continue the decompilation at one of these successors.

To illustrate the latter situation, consider a static method call `C.m()` inside the `try` of a `try-catch` block and its compilation to assembly (shown in Fig. 5). The Locals decompiler, and several decompilers after it, produce one successor abstract state after the call to `C.m()` (line 2). In order to soundly analyze a possible `throw` in `C.m()`, the decompiler that handles exceptions must add one more successor at the method call for the `catch` block at $\mathrm{L}_{\mathrm{catch}}$. The challenge is to generate appropriate low-level abstract states for the successor at $\mathrm{L}_{\mathrm{catch}}$. For example, the exceptions decompiler might want to direct all other decompilers to transform their abstract states before the static method call and produce an abstract state for $\mathrm{L}_{\mathrm{catch}}$ from it by clobbering certain registers and portions of memory.

The mechanism we propose is based on the observation that we already have a pipeline of decompilers that is able to transform the abstract states at all levels when given a sequence of machine instructions. To take advantage of this we require a decompiler to provide, for each newly introduced successor, a list of machine instructions that will be "run" through the decompilation pipeline (using `step`) to produce the missing lower-level abstract states. To achieve this, we extend the `succ` type (used in the return of `step`) to also carry an optional list of *machine* instructions (of type $\texttt{instr}_C$):

$$\textbf{type}\ \texttt{succ} = \texttt{loc} \times (\texttt{abs} \times ((\texttt{instr}_C\ \texttt{list})\ \texttt{option}))\ . \qquad \boxed{\textsc{Decompiler}}$$

As a side-condition, the concrete machine instructions returned by `step` should not include control-flow instructions (e.g., `jump`). We also extend the concrete machine instruction set with instructions for abstracting effects; for example, there is a way to express that register $\mathbf{r}_x$ gets modified arbitrarily (`havoc` $\mathbf{r}_x$).[1]

[1] Such instructions are also useful for abstracting x86 instructions for which we currently do not handle.

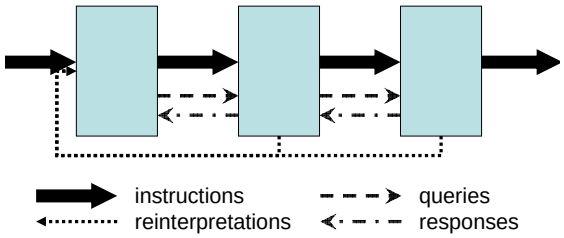

Fig. 6. Communication between decompilers. The primary communication channel is the instruction stream, but queries and reinterpretations provide means for higher-level decompilers to communicate information to lower-level decompilers.

Both queries and these *reinterpretations* introduce a channel of communication from higher-level decompilers to lower-level ones, but they serve complimentary purposes. For one, reinterpretations are initiated by high-level decompilers, while queries are initiated by low-level decompilers. We want to use queries when we want the question to be decompiled, while we prefer to communicate through reinterpretations when we want the answers to be decompiled. Fig. 6 summarizes these points. In Appendix A, we give the product decompiler that ties decompilers together (with queries and reinterpretations).

Soundness of Decompiler Pipeline. One of the main advantages of the modular architecture we describe in this paper is that we can modularize the soundness argument itself. This modularization increases the trustworthiness of the program analysis and is a first step towards generating machine-checkable proofs of soundness, in the style of Foundational Proof-Carrying Code [App01].

Since we build on the framework of abstract interpretation, the proof obligations for demonstrating the soundness of a decompiler are fairly standard local criteria, which we sketch here. Soundness of a decompiler module is shown with respect to the semantics of its input and output languages given by concrete transition relations. In particular, leaving the program implicit, we write $I_{\mathcal{L}} : l \rightsquigarrow_{\mathcal{L}} l'@\ell$ for the one-step transition relation of the input (lower-level) machine, which says that on instruction $I_{\mathcal{L}}$ and pre-state l, the post-state is l' at program location ℓ (similarly for the output (higher-level) machine $\mathcal{H}$). As usual, we can specify whatever safety policy of interest by disallowing transitions that would violate the policy (i.e., modeling errors as "getting stuck"). Also, as usual, we need to define a *soundness relation* $l \precsim a$ between concrete states for the input machine and abstract states, as well as a *simulation relation* $l \sim h$ between concrete states of the input and output machines.

Note that for a given assembly program, we use the same locations for all decompilations since we consider one-to-one decompilations for presentation purposes (otherwise, we would consider a correspondence between locations at different levels). Let $\mathcal{L}_0$ and $\mathcal{H}_0$ denote the initial machine states (as a mapping from starting locations to states) such that they have the same starting locations

each with compatible states (i.e., $\mathrm{dom}(\mathcal{L}_0) = \mathrm{dom}(\mathcal{H}_0)$ and $\mathcal{L}_0(\ell) \sim \mathcal{H}_0(\ell)$ for all $\ell \in \mathrm{dom}(\mathcal{L}_0)$). Now consider running the decompiler pipeline to completion (i.e., to fixed point) and let $\mathcal{A}_{\text{INV}}$ be the mapping from locations to abstract states at fixed point. Note that $\mathcal{A}_{\text{INV}}$ must contain initial abstract states compatible with concrete states in $\mathcal{L}_0$ (i.e., $\mathrm{dom}(\mathcal{A}_{\text{INV}}) \subseteq \mathrm{dom}(\mathcal{L}_0)$ and $\mathcal{L}_0(\ell) \precsim \mathcal{A}_{\text{INV}}(\ell)$ for all $\ell \in \mathrm{dom}(\mathcal{L}_0)$).

We can now state the local soundness properties for a decompiler module's `step`. A decompiler's `step` need only give sound results when the query object it receives as input yields answers that are sound approximations of the machine state, which we write as $h \gtrapprox q$ (and which would be defined and shown separately).

Property 1 (Progress). If $l \sim h$, $l \precsim a$, $h \gtrapprox q$, $\texttt{step}((q, a), I_{\mathcal{L}}) = (I_{\mathcal{H}}, A')$ and $I_{\mathcal{H}} \mathbin{⫶} h \rightsquigarrow_{\mathcal{H}} h'@\ell$, then $I_{\mathcal{L}} \mathbin{⫶} l \rightsquigarrow_{\mathcal{L}} l'@\ell$ (for some h').

Progress says that whenever the decompiler can make a step *and* whenever the output machine is not stuck, then the input is also not stuck. That is, a decompiler residuates soundness obligations to higher-level decompilers through its output instruction. Thus far, we have not discussed the semantics of the intermediate languages very precisely, but here is where it becomes important. For example, for stack slots to be soundly translated to temporaries by the Locals decompiler, the semantics of the memory write instruction in Locals IL is not the same as a memory write in the assembly in that it must disallow updating such stack regions. To implement a decompiler pipeline that enforces a particular safety policy encoded in the concrete machine, we could have a module at the end that simply checks syntactically that all the "possibly unsafe" instructions have been decompiled away (e.g., for memory safety, all memory read instructions have been decompiled into various safe read instructions).

Property 2 (Preservation). If $l \sim h$, $l \precsim a$, $h \gtrapprox q$ and $\texttt{step}((q, a), I_{\mathcal{L}}) = (I_{\mathcal{H}}, A')$, then for every l' such that $I_{\mathcal{L}} \mathbin{⫶} l \rightsquigarrow_{\mathcal{L}} l'@\ell$, there exists h', a' such that $I_{\mathcal{H}} \mathbin{⫶} h \rightsquigarrow_{\mathcal{H}} h'@\ell$ where $l' \sim h'$ and $a' = \mathcal{A}_{\text{INV}}(\ell)$ where $l' \precsim a'$.

Preservation guarantees that for every transition made by the input machine, the output machine simulates it and the concrete successor state matches one of the abstract successors computed by `step` (in $\mathcal{A}_{\text{INV}}$).

3 Decompiler Examples

In this section, we describe a few decompilers from Fig. 1. While each individual decompiler is relatively straightforward, together they are able to handle the complexities typically associated with analyzing assembly code. Each decompiler has roughly the same structure. For each one, the input instruction language is given by the lower-level decompiler in the pipeline. Each decompiler defines a type of output instructions `instr` for expressing the result of decompilation and a notion of abstract state `abs`. The abstract state generally is a mapping from

variables to *abstract values*, though the kinds of variables may change through the decompilation.

For each decompiler, we give the instruction of the output language, the lattice of abstract values, and a description of the decompilation function `step`. We use the simplified notation $\texttt{step}(a_{curr}, I_{in}) = (I_{out}, a_{succ})$ to say that in the abstract state a_{curr} the instruction I_{in} is decompiled to I_{out} and yields a successor state a_{succ}. We write $a_{succ}@\ell$ to indicate the location of the successor, but we elide the location in the common case when it is "fall-through". A missing successor state a_{succ} means that the current analysis path ends. We leave the query object implicit, using q to stand for it when necessary. Since each decompiler has similar structure, we use subscripts with names of decompilers or languages when necessary to clarify to which module something belongs.

Decompiling Calls and Locals. The Locals module deals with stack conventions and introduces the notion of statically-scoped local variables. The two major changes from assembly instructions (I_C) are that call and return instructions have actual arguments.

$$\texttt{instr} \quad I_L ::= I_C \mid x := \texttt{call}\ \ell(e_1, ..., e_n) \mid x := \texttt{icall}\ [e](e_1, ..., e_n) \mid \texttt{return}\ e$$

The abstract state L includes a mapping Γ from variables x to abstract values τ, along with two additional integers, n_{lo} and n_{hi}, that delimit the current activation record (i.e., the extent of the known valid stack addresses for this function) with respect to the value of the stack pointer on entry.

$$\texttt{abs} \quad L ::= \langle \Gamma; n_{lo}; n_{hi} \rangle$$

The variables mapped by the abstract state include all machine registers and variables $\mathbf{t}_n$ that correspond to stack slots (with the subscript indicating the stack offset of the slot in question). The abstract values are defined by the following grammar:

$$\text{abstract values} \quad \tau ::= \top \mid n \mid \mathsf{sp}(n) \mid \mathsf{ra} \mid \&\ell \mid \mathsf{cs}(r)$$

We need only track a few abstract values τ: the value of stack pointers $\mathsf{sp}(n)$, the return address for the function ra, code addresses for function return addresses $\&\ell$, and the value of callee-save registers on function entry $\mathsf{cs}(r)$. These values form a flat lattice, with the usual ordering ($\top$ being the top element).

Many of the cases for the `step` function propagate the input instruction unchanged and update the abstract state. We show below the definition of `step` for the decompilation of a stack memory read to a move from a variable. For simplicity, we assume here that all stack slots are used for locals. This setup can be extended to allow higher-level decompilers to indicate (through some high-to-low communication) which portions of the stack frame it wants to handle separately.

$$\frac{\Gamma \vdash e : \mathsf{sp}(n) \quad n_{lo} \le n \le n_{hi} \quad n \equiv 0 \pmod 4}{\texttt{step}(\langle \Gamma; n_{lo}; n_{hi} \rangle, r := \mathbf{m}[e]) \ = \ (r := \mathbf{t}_n, \langle \Gamma[r \mapsto \Gamma(\mathbf{t}_n)]; n_{lo}; n_{hi} \rangle)}$$

We write $\Gamma \vdash e : \tau$ to say that in the abstract state $\langle \Gamma; n_{lo}; n_{hi} \rangle$, the expression e has abstract value τ. The first premise identifies the address as a stack address, the second checks that the address is within the activation record, while the last ensures the address is word-aligned. Again for simplicity in presentation, we only consider word-sized variables here. For verifying memory safety, a key observation is that Locals proves once and for all that such a read is to a valid memory address; by decompiling to a move instruction, no higher-level decompiler needs to do this reasoning. The analogous translation for stack writes appears on, for example, line 19 in Fig. 3.

The following rule gives the translation of function calls:

$$\frac{\begin{array}{l}\Gamma(x_{ra}) = \&\ell \quad \Gamma(\mathbf{r}_{sp}) = \mathtt{sp}(n) \quad q.\mathtt{isFunc}(e) = k \quad n \equiv 0 \pmod 4 \\ \Gamma' = \mathit{scramble}(\Gamma, n, k)\end{array}}{\begin{array}{l}\mathtt{step}(\langle \Gamma; n_{lo}; n_{hi} \rangle, \mathtt{icall}\ [e]) \\ \qquad = (x_{rv} := \mathtt{icall}\ [e](x_1, ..., x_k), \langle \Gamma'[\mathbf{r}_{sp} \mapsto \mathtt{sp}(n{+}4)]; n_{lo}; n_{hi} \rangle @ \ell)\end{array}}$$

It checks that the return address is set, $\mathbf{r}_{sp}$ contains a word-aligned stack pointer and is word-aligned, and is a call according to the query. Based on the calling convention and number of arguments, it constructs the call with arguments and the return register. The successor state Γ' is obtained first by clearing any non-callee-save registers and temporaries corresponding to stack slots in the callee's activation record, which is determined by *scramble* using the calling convention and n and k. Then, $\mathbf{r}_{sp}$ is updated, shown here according to the x86 calling convention where the callee pops the return address. In implementation, we parameterize by a description of the calling convention. There is also an additional decompilation case for direct calls, which is analogous to indirect calls.

On a `return`, the Locals decompiler checks that the stack pointer is reset correctly and the callee-save registers have been restored. It then re-writes the `return` to include the return value register.

$$\frac{\Gamma(\mathbf{r}_{sp}) = \mathtt{sp}(4) \quad \Gamma(r) = \mathtt{cs}(r) \ \text{(for all callee-save registers } r)}{\mathtt{step}(\langle \Gamma; n_{lo}; n_{hi} \rangle, \mathtt{return}) \ = \ \mathtt{return}\ x_{rv}}$$

Stack Overflow Checking. An interesting aspect of the Locals decompiler is that it is designed to reason about stack overflow. This handling is mandatory for eventually proving its soundness. There are many possibilities for detecting stack overflow. Our implementation is for code compiled with `gcc`'s (and `gcj`'s) built-in mechanism for detecting stack overflow (`-fstack-check`). This mechanism relies on a inaccessible guard page and on stack probes inserted by the compiler to ensure that no function can accidentally skip over the guard page.

We show below a transition rule that identifies a stack probe and extends n_{lo} in the abstract state:

$$\frac{\Gamma \vdash e_1 : \mathtt{sp}(n) \quad n_{lo} - \mathtt{GUARD\_PAGE\_SIZE} \leq n < n_{lo}}{\mathtt{step}(\langle \Gamma; n_{lo}; n_{hi} \rangle, \mathbf{m}[e_1] := e_2) \ = \ (\mathtt{nop}, \langle \Gamma; n; n_{hi} \rangle)}$$

A probe is a stack access that is below the current n_{lo} but must be within the size of the guard page (`GUARD_PAGE_SIZE`). Such an access either aborts the program safely, or $\mathsf{sp}(n)$ is a valid stack address, so n can be used as the new n_{lo}. To our knowledge, this mechanism has not been previously formalized for the purpose of verifying the absence of stack overflow.

Symbolic Evaluator. The SymEval ($\mathcal{E}$) module does the following analysis and transformations for higher-level decompilers to resolve some particularly pervasive problems when analyzing assembly code.

1. *Simplified and Normalized Expressions.* High-level operations get compiled into long sequences of assembly instructions with intermediate values exposed (as exemplified in Fig. 3), as a direct result of three-address code. To analyze one instruction at a time, we need to assign types to all intermediate expressions, but this undertaking quickly becomes unwieldy. Additionally, arithmetic equivalences are used extensively by compilers (particularly in optimized code). We want to accumulate larger expression trees and perform arithmetic simplification and normalization before assigning types. Observe how SymEval does this work in the example decompilation of line 17 in Fig. 4.
2. *Static Single Assignment (SSA).* In contrast to source-level variables, flow-sensitivity is generally required to analyze registers because registers are reused for unrelated purposes. To have a set variables suitable for source-level analyses, the symbolic evaluator yields an SSA-like (or functional-like) program representation.
3. *Global Value Numbering (GVN).* The same variable may also be placed in multiple locations (yielding an equality on those locations). For example, to check that a reference stored on the stack is non-null, a compiler must emit code that first loads it into a register. On the non-null path, an assembly-level analysis needs to know that the contents of both the register and the stack slot is non-null. So that higher-level decompilers do not have to deal with such low-level details, the symbolic evaluator presents a single *symbolic value* α that abstracts some unknown value but is stored in both the register and the stack slot (implicitly conveying the equality). Combined with the above, the symbolic evaluator can be viewed as implementing an extended form of GVN [AWZ88,GN04].

These issues with analyzing assembly were identified in our prior work [CCNS05], but we describe here a way to modularize those techniques via decompilation.

The output instruction language for the SymEval decompiler is essentially the same as that for Locals, except that the operands may contain expression trees. However, the expression language extends the input expressions (i.e., of LocalsIL) with symbolic values α and memory read expressions.

`expr`	$e_{\mathcal{E}} ::= \alpha \mid \mathbf{m}[e_{\mathcal{E}}] \mid \cdots$
symbolic values	α, β

Note that the memory read expression leaves the memory from which the value is read implicit (i.e., reads are always with respect to the current memory). (Including memory read expressions in symbolic evaluation is actually an extension to the ideas described in our prior work [CCNS05].)

The abstract state consists of a finite map Σ from variables x to expressions in its output language $e_{\mathcal{E}}$. To summarize what we track, the concretization of the abstract state $E = \langle x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n \rangle$ is

$$(\exists \alpha_1, \alpha_2, \ldots, \alpha_m.\ \ x_1 = e_1 \wedge x_2 = e_2 \wedge \cdots \wedge x_n = e_n)$$

where $\alpha_1, \alpha_2, \ldots, \alpha_m$ are the symbolic values that appear in E.

We write $e \Downarrow e'$ for the normalization of expression e to e'. The details of the normalization are not particularly relevant, except that we require the following correctness condition: two expressions e_1 and e_2 normalize to syntactically equal expressions (i.e., $e_1 \Downarrow e$ and $e_2 \Downarrow e$) only if e_1 and e_2 are semantically equivalent. Conversely, the precision of normalization determines the equalities we can infer.

With an accumulated value state Σ, the decompilation of instructions is straightforward. For each input expression $e_{\mathcal{L}}$, we substitute for the registers and temporaries the accumulated expression $e_{\mathcal{E}}$ for them in Σ and normalize; assignments are then replaced by bindings of a fresh symbolic value. For example, the decompilation of an `icall` is as follows:

$$\frac{\Sigma(e_0) \Downarrow e'_0 \quad \cdots \quad \Sigma(e_n) \Downarrow e'_n \quad (\alpha \text{ fresh})}{\texttt{step}(\Sigma, x := \texttt{icall}\ [e_0](e_1, ..., e_n)) \ =\ (\alpha = \texttt{icall}\ [e'_0](e'_1, ..., e'_n), \Sigma')}$$

where Σ' is the value state that reflects the effects of the call (as described by Locals), for instance, the register state is scrambled except for the callee-save registers. In the above, we treat Σ as a substitution, writing $\Sigma(e)$ for the expression where registers and temporaries are replaced by their mapping in Σ. Compare this rule with the example decompilation shown on line 17 in Fig. 4.

To accumulate, for instance, a memory read, we have the following rule:

$$\frac{\Sigma(e) \Downarrow e' \quad (\alpha \text{ fresh})}{\texttt{step}(\Sigma, x := \mathbf{m}[e]) \ =\ (\alpha = \mathbf{m}[e'], \Sigma[x \mapsto \mathbf{m}[e']])}$$

Since we compute the normalization for decompilation anyway, we always keep normalized expressions in the value state. However, not shown here is that we memoize the assignment of symbolic values to expressions, so equivalent expressions (as determined by normalization) are assigned the same symbolic value. In the case where the value of this memory read has already been assigned a symbolic value, then we can omit the output instruction.

Because memory reads expressions are always in terms of the current memory, on a write to memory, we forget all such expressions and replace them by symbolic values. This conservative (and simple) modeling of the heap has been sufficient because while the structure of the memory read is lost on a write, a "handle" to its value is preserved as a symbolic value. To strengthen this

modeling, one could imagine using a theorem prover or querying to try to obtain disaliasing information (i.e., $e_1 \neq e_2$) in order to preserve some memory read expressions, or introducing explicit heap variables in read expressions (e.g., $\mathsf{sel}(m, e)$) to further postpone alias analysis. However, both these solutions seem heavyweight compared with having higher-level decompilers strengthen their type systems to keep whatever necessary shape information about past heaps it needs.

Both for the symbolic evaluator and higher-level decompilers, we keep with each subexpression, a symbolic value that denotes it. With this information, we inductively define an operation $\uparrow\cdot$ over expressions that drops memory reads:

$$\uparrow(\mathbf{m}[e])_\alpha \stackrel{\text{def}}{=} \alpha \quad \uparrow n \stackrel{\text{def}}{=} n \quad \uparrow(e_1 + e_2) \stackrel{\text{def}}{=} \uparrow e_1 + \uparrow e_2 \quad \ldots$$

where the subscript on expressions indicates the symbolic value that is assigned to it. Lifting this operation to value states, we define the transition on memory writes:

$$\frac{\Sigma(e_1) \Downarrow e_1' \quad \Sigma(e_2) \Downarrow e_2'}{\mathtt{step}(\Sigma, \mathbf{m}[e_1] := e_2) \;=\; (\mathbf{m}[e_1'] := e_2', \uparrow\Sigma)}$$

Widen and Ordering. Recall that a value state represents a (finite) conjunction of equality constraints (of a particular form). Symbolic values essentially provide names for equivalence classes of expressions. To widen value states, we treat expressions as uninterpreted functions. Then, the widen algorithm is essentially the upper bound operation described in our previous work [CCNS05], which is a special case of the join for the general theory of uninterpreted functions [GTN04,CL05]. However, special care must be taken to handle memory read expressions correctly for both widening and ordering. Read expressions cannot be compared between abstract states because their memory state may be different (and because memory is implicit, we cannot tell otherwise). Therefore, we must also forget all memory read expressions at join points in the control-flow graph.

To compute the non-trivial widening $\Sigma_1 \triangledown \Sigma_2$, we first need to forget all memory read expressions. Let Σ denote the result of the widen. The resulting value state's domain should be the intersection of the domain of Σ_1 and Σ_2 with mappings to expressions that preserve only syntactically equivalent structure. For the moment, let us denote an expression in the resulting state as the corresponding pair of expressions in the input states. That is, the resulting state is defined as

$$\Sigma(x) \stackrel{\text{def}}{=} \big\langle (\uparrow\Sigma_1)(x), (\uparrow\Sigma_2)(x) \big\rangle \,.$$

Then, we translate these pairs recursively over the structure of expressions to yield the resulting value state. For expressions e_1 and e_2, if their structures do not match, then they are abstracted as a fresh symbolic value. Formally, let $\ulcorner\cdot\urcorner$

be the translation of these pairs to a single expression:

$$\begin{array}{ll}
\ulcorner\langle\alpha_1, \alpha_2\rangle\urcorner \stackrel{\text{def}}{=} \beta & \text{where } \beta \text{ fresh} \\
\ulcorner\langle n, n\rangle\urcorner \stackrel{\text{def}}{=} n & \\
\ulcorner\langle e_1 + e_1', e_2 + e_2'\rangle\urcorner \stackrel{\text{def}}{=} \ulcorner\langle e_1, e_2\rangle\urcorner + \ulcorner\langle e_1', e_2'\rangle\urcorner & \\
\quad \ldots & \\
\ulcorner\langle e_1, e_2\rangle\urcorner \stackrel{\text{def}}{=} \beta & \text{where } \beta \text{ fresh} \qquad \text{(otherwise)}
\end{array}$$

Note that each distinct pair of symbolic values maps to a fresh symbolic value. Soundness of this operation follows from the join of uninterpreted functions (see [GTN04,CL05]).

The ordering on abstract states is essentially given by implication of the equality constraints; however, we can only compare values states without memory read expressions. Let us first consider deciding ordering on read-free value states $\Sigma_1 \dot{\sqsubseteq} \Sigma_2$. That is, we want to decide whether $\gamma(\Sigma_1) \Rightarrow \gamma(\Sigma_2)$ where γ denotes the concretization function (as standard). We consider the pairs of symbolic values $\langle\alpha_1, \alpha_2\rangle$ that result from the analogous procedure as for the widen. Recall that symbolic values name equivalence classes of expressions. If α_2 is in only one pair, then the equalities implied by Σ_2 named by α_2 are also implied by Σ_1 (named by α_1), so if all α_2's (i.e., right projections) appear in at most one pair, then all the equalities implied by Σ_2 are also implied by Σ_1. This observation gives an algorithm for deciding $\Sigma_1 \dot{\sqsubseteq} \Sigma_2$. Now, consider deciding $\Sigma_1 \sqsubseteq \Sigma_2$ for arbitrary values states. It is clear that $\gamma(\Sigma) \Rightarrow \gamma(\uparrow\Sigma)$ for all Σ, so we can say

$$\Sigma_1 \sqsubseteq \Sigma_2 \quad \text{if} \quad \uparrow\Sigma_1 \dot{\sqsubseteq} \uparrow\Sigma_2 \text{ and } \Sigma_2 = \uparrow\Sigma_2 \,.$$

The above widening operator has the stabilizing property because (1) the first state has a finite number of expressions; (2) dropping memory read expressions can occur at most once; and (3) each iteration can only partition existing equivalence classes. With a finite number of expressions, there can only be a finite number of equivalence classes. It is worthwhile to point out why this upper bound operation is simpler and does not require additional heuristics to ensure stabilization as compared to the join for the general theory of uninterpreted functions [GTN04,CL05]. The key observation is that we never push assumptions that union equivalence classes (e.g., to reflect a branch condition). This restriction prevents cyclic dependencies, that is, a constraint $\alpha = e$ where e contains α.

Decompiling Object-Oriented Features. The OO decompiler (O) recognizes compilations of class-based object-oriented languages, such as Java and C#. These core object-oriented features are generally compiled in the same way: an object is a record that contains a pointer to its virtual dispatch table and its fields. The dispatch table is then a record that contains pointers to its methods. Therefore, OO can identify virtual method dispatch, field reads, and field writes based on this compilation strategy. In this section, we describe OO specialized

to object layout used by `gcj`; in our implementation, it is parameterized by an object layout description.

The output instruction language for the OO decompiler includes the instructions from the symbolic evaluator, except it is extended for virtual method dispatch, field reads, and field writes:

$$\begin{array}{ll} \texttt{instr} & I_O ::= I_E \mid \alpha = \texttt{putfield}\ [e, n] \mid \alpha = \texttt{invokevirtual}\ [e_0, n](e_1, ..., e_n) \\ \texttt{expr} & e_O ::= e_E \mid \texttt{getfield}\ [e, n] \end{array}$$

Almost all of the heavy lifting has been done by the symbolic evaluator, so OO is quite simple. The abstract values that we need to track are straightforward: a type for object references, which may be qualified as non-null or possibly null. However, the variables for OO are *symbolic values* instead of machine state elements.

$$\begin{array}{ll} \texttt{abs} & \Gamma ::= \cdot \mid \Gamma, \alpha : \tau \\ \text{types} & \tau ::= \top \mid [\mathsf{nonnull}]\ \mathsf{obj} \end{array}$$

Typing is also straightforward, except types of fields are obtained through queries.

The decompilation of virtual method dispatch (as on line 17 in Fig. 4) is as follows:

$$\frac{\begin{array}{l}\Gamma(\beta) = \mathsf{nonnull\ obj} \qquad \Gamma \vdash e_1 : \tau_1\ \cdots\ \Gamma \vdash e_m : \tau_m \\ q.\texttt{isMethod}(\beta, n) = \tau_1 \times \cdots \times \tau_m \to \tau\end{array}}{\begin{array}{l}\texttt{step}(\Gamma, \alpha = \texttt{icall}\ [\mathbf{m}[\mathbf{m}[\beta] + n]](\beta, e_1, ..., e_m)) \\ \qquad\qquad = (\alpha = \texttt{invokevirtual}\ [\beta, n](e_1, ..., e_m), \Gamma[\alpha \mapsto \tau])\end{array}}$$

It checks that the object reference is non-null and that the dispatch table is obtained from the same object as the object being passed as the receiver object. Observe that since the abstract state is independent of the register and memory state, the successor abstract state is particularly easy to derive. Decompilations for field reads (`getfield`) and field writes (`putfield`) are similar. Note that the symbolic evaluator enables the use of such simple types and rules, as opposed to the dependent types used in our prior work [CCNS05] (though we may still choose to extend the type system to include such dependent types for dispatch table and methods if this information needs to be tracked across writes or join points).

One additional bit of interesting work is that it must recognize null-checks and strengthen an obj to a nonnull obj. For example,

$$\frac{I = \texttt{if}\ (e_\alpha = 0)\ \ell \quad \Gamma \vdash e_\alpha : \mathsf{obj}}{\texttt{step}(\Gamma, I)\ =\ (I, \Gamma[\alpha \mapsto \mathsf{nonnull\ obj}])}$$

Because of the symbolic evaluator, OO simply updates the type of a symbolic value α and need not worry about the equivalences between all the registers or temporaries that contain α.

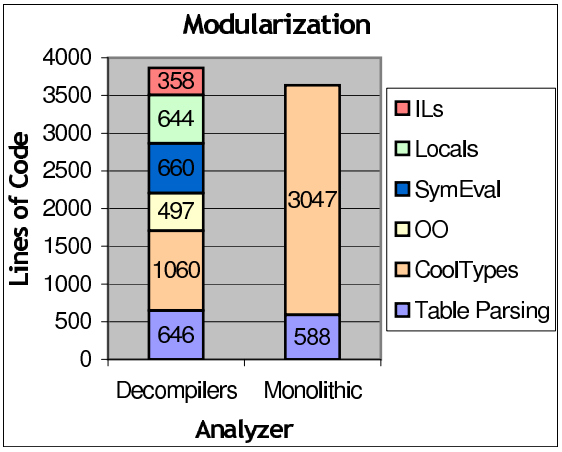

Fig. 7. Size of decompiler modules for the `coolc` pipeline (Decompilers) compared with our previous monolithic assembly-level analyzer (Monolithic).

Java Type Inference. After the decompilations performed by OO, we get a language roughly-like the JVML but a bit higher-level—there is no operand stack, but rather symbolic values, which are closer to source-level variables. The JavaTypes decompiler introduces the Java source types. It obtains the class hierarchy information and answers the queries of lower-level decompilers with it. With the class hierarchy, the analysis it performs is exceedingly simple and well-studied—somewhere between Java bytecode verification and Java type-checking in complexity. The only place where flow-sensitivity is needed is to handle down casts (which is like the null-check in OO). In our implementation, most of the work in JavaTypes is actually not in the analysis, but rather obtaining the class hierarchy. We obtain the class hierarchy by reading tables in the data segment generated by `gcj` that are used to implement reflection, and so we do not require any additional annotations to recover types.

Implementation and Experience. We have implemented and tested the above decompiler modules in multiple decompiler pipelines, including three main ones for assembly generated from: Java programs by `gcj`, C programs by `gcc`, and Cool programs by `coolc`. All decompiler pipelines start from a very simple untyped RISC-like assembly language to minimize architecture dependence. We have parsers for x86 and MIPS that translate to this generic assembly. The Locals module is parameterized by the calling convention, so we can easily handle several different calling conventions (e.g., standard x86, standard MIPS, or the non-standard one used by `coolc`).

Each of the decompiler modules described above is actually quite small (at most ∼600 lines of OCaml). Furthermore, each module is approximately the same size providing some evidence for a good division of labor. The overhead (i.e., the definition of the intermediate languages and associated utility func-

tions) seems reasonable, as each language only required 100–150 lines of OCaml. The entire `coolc` pipeline (including the Cool type analysis but not the framework code) is 3,865 lines compared to 3,635 lines for a monolithic assembly-level analyzer from our previous work [CCNS05], which uses the classic reduced product approach. Cool is a fairly realistic subset of Java, including features such as exceptions, so the CoolTypes module includes the handling of exceptions as described in Sec. 2. The additional code is essentially in the definition of the intermediate languages, so what we conclude is that our pipeline approach does give us a modular and easier to maintain design without imposing an unreasonable code size penalty with respect to the monolithic version. These results are shown in Fig. 7, and we can observe visually that the decompiler modules are indeed approximately equal in size. Additionally, note that 2,159 and 1,515 of the 3,865 lines of the `coolc` decompiler pipeline are reused as-is in the `gcj` and `gcc` pipelines, respectively.

Comparing the implementation experience with our previous assembly-level analyzer, we found that the separation of concerns imposed by this framework made it much easier to reason about and implement such assembly-level analyses. For example, because of the decompilations, Cool/Java type inference is no longer intermingled with the analysis of compiler-specific run-time structures. With this framework, we also obtained comparable stability in a much shorter amount of time. Many of the bugs in the implementation described in our prior work [CCNS05] were caused by subtle interactions in somewhat ad-hoc modularization there, which simply did not materialize here. As an example of the utility of this approach, after the implementation for the class table parser was complete, one of the authors was able to implement a basic Java type inference module in 3–4 hours and ∼500 lines of code (without the handling of interfaces and exceptions).

4 Case Studies

To explore the feasibility of applying existing source-level tools to assembly code, we have used BLAST [HJM+02] (a model checker for C) and Cqual [FTA02] (a type qualifier inference for C) on decompilations produced by our `gcc` pipeline. To interface with these tools, we have a module that emits C from SymEval IL (though ideally we might prefer to go directly to the tools internal representation to avoid dealing with the idiosyncrasies of C front-ends). SymEval IL is essentially C, as register reuse with unrelated types have been eliminated by SSA and expression trees have been recovered. However, while a straightforward translation from SymEval IL produces a valid C program that can be (re)compiled and executed, the typing is often too weak for source-level analysis tools. To avoid this issue for these experiments, we use debugging information to recover types. When debugging information is not available, we might be able to obtain typing information using a decompiler module that implements a type reconstruction algorithm such as Mycroft's [Myc99].

Test Case		Code Size C (loc)	Code Size x86 (loc)	Decomp. (sec)	Verification Orig. (sec)	Verification Decomp. (sec)
`qpmouse.c`	(B)	7994	1851	0.74	0.34	1.26
`tlan.c`	(B)	10909	10734	8.16	41.20	94.30
`gamma_dma.c`	(Q)	11239	5235	2.44	0.97	1.05

Table 1. Decompilation and verification times using BLAST (B) and Cqual (Q).

We have taken the benchmarks shown in Table 1, compiled them to x86 (unoptimized), and decompiled them back to C before feeding the decompilations to the source-level tools (B for BLAST and Q for Cqual). In all cases, we checked that the tools could verify the presence (or absence) of bugs just as they had for the original C program. In the table, we show our decompilation times and the verification times of both the original and decompiled programs on a 1.7GHz Pentium 4 with 1GB RAM. The BLAST cases `qpmouse.c` and `tlan.c` are previously reported Linux device drivers for which BLAST checks that `lock` and `unlock` are used correctly [HJM+02]. For `gamma_dma.c`, a file from version 2.4.23 of the Linux kernel, Cqual is able to find in the decompiled program a previously reported bug involving the unsafe dereference of a user-mode pointer [JW04]. Both Cqual and BLAST require interprocedural analyses and some C type information to check their respective properties. We have also repeated some of these experiments with optimized code. With `qpmouse`, we were able to use all the `-O2` optimizations in `gcc` 3.4.4, such as instruction scheduling, except `-fmerge-constants`, which yields code that reads a byte directly from the middle of a word-sized field, and `-foptimize-sibling-calls`, which introduces tail calls. The latter problem we could probably handle with an improved `Locals` module, but former is more difficult. One limitation we have observed with using the debugging information is that complicated pointer offsets are challenging to map back to C `struct` accesses. Also, we do not yet handle all assembly instructions, particularly kernel instructions.

5 Related Work

Combinations of Analyses. In abstract interpretation, the problem of combining abstract domains has also been considered by many. Cousot and Cousot [CC79] define the notion of a *reduced product*, which gives a "gold standard" for precise combinations of abstract domains. In contrast to the direct product (i.e., a Cartesian product of independent analyses), obtaining a reduced product implementation is not automatic; they generally require manual definitions of reduction operators, which depend on the specifics of the domains being combined (e.g., [CMB+95]). Roughly speaking, we propose a framework for building reduced products based on decompilation, which is particular amiable for

modularizing the analysis of assembly code. Cortesi *et al.* [CCH94] describe a framework (called an *open product*) that takes queries as the central (and only) means of communication. They allow arbitrary queries between any pair of domains, whereas our queries are more structured through decompilation. With this structure, modules need only agree upon a communication interface with its neighbors (i.e., the decompiler immediately below it and the one immediately above it). An alternative framework for combination of abstract domains fixes one common language for communication (e.g., first-order logic), for example, as in Chang and Leino [CL05]. In that framework, which owes inspiration to the Nelson-Oppen combination of decision procedures [NO79], there is a centralized dispatcher (the *congruence-closure domain*), whereas we have a more distributed communication model.

Combining program analyses for compiler optimization is a well-known and well-studied problem. It is widely understood that optimizations can lead to mutually beneficial interactions, which leads to a *phase ordering* problem. At the same time, it is known that manually constructed combinations of analyses can be more precise than an iterative application of individual optimizations but at the cost of modularity. Lerner *et al.* [LGC02] propose modular combinations of compiler optimizations also by integrating analysis with program transformation, which then serve as the primary channel of communication between analyses. We, however, use transformation for abstraction rather than optimization. For this reason, we use layers of intermediate languages instead of one common language, which is especially useful to allow residuation of soundness obligations. They also found it necessary to have a side-channel for communicating facts contained in abstract states (*snooping*), which is similar to our query mechanism except that we motivate making queries a first-class mechanism.

Decompilation. Practically all analysis frameworks, particularly for low-level code, perform some decompilation or canonicalization for client analyses. For example, the Soot framework [VRCG+99] for the JVML provides several different levels intermediate representations (Baf, Jimple, Shimple, and Grimp) that assign types, decompile exceptions, convert into SSA, and introduce tree-structured expressions. Our resulting decompilations for Java are similar to theirs, though there are variances driven by differences in focus. They want to support optimization, while we are more concerned with verification. This difference shows up in, for example, we convert into SSA and introduce tree-structured expressions much earlier in our pipeline. Of course, we have also been concerned with a framework that allows additional pipelines for different languages to be built quickly and easily, as well as starting from assembly code.

Similarly, CodeSurfer/x86 [BR04], which is built on IDAPro [IDA] and CodeSurfer [AZ05], seeks to provide a higher-level intermediate representation for analyzing x86 machine code. At the core of CodeSurfer/x86 is a nice combined integer and pointer analysis (*value set analysis*) for abstract locations, which may be machine registers, stack slots, or `malloc` sites. The motivation for this analysis is similar to that for the Locals module, except we prefer to handle the heap separately in language-specific ways. Their overall approach is a bit

different from ours in that they try to decompile without the assistance of any higher-level language-specific analysis, which leads to complexity and possible unsoundness in the handling of, for example, indirect jumps and stack-allocated arrays. While even they must make the assumption that the code conforms to a "standard compilation model" where a run-time stack of activation records are pushed and popped on function call and return, their approach is more generic out of the box. We instead advocate a clean modularization to enable reuse of decompiler components in order to make customized pipelines more palatable.

Tröger and Cifuentes [TC02] give a technique to identify virtual method dispatch in machine code binaries based on computing a backward slice from the indirect call. They also try to be generic to any compiler, which necessarily leads to difficulties and imprecision that are not problems for us.

Cifuentes *et al.* [CSF98] describe a decompiler from SPARC assembly to C. Driven by the program understanding application, most of their focus is on recovering structured control-flow, which is often unnecessary (if not undesirable) for targeting program analyses. Mycroft [Myc99] presents an algorithm for recovering C types (including recursive data types) based on a variant of Milner's unification-based type inference algorithm [Mil78]. We could also use this technique in a decompiler module to recover recursive data types. By building it on top of the Locals module (as well as SymEval), it is possible we could enrich the results that can be currently obtained by this technique.

Reusing Source-Level Analyses. Rival [Riv03] shows how to use debugging information and the invariants obtained by a source-level analysis to verify that they hold for the compilation to assembly. Unfortunately, this verification process still requires implementing a corresponding assembly-level analysis with all the complications we have described. One advantage of their approach is that the verification can be done in a linear scan by using the translated invariants at control-flow join points. This separation could be beneficial because it may be more efficient to compute the fixpoint at the source-level or for the mobile-code application where the checking on the consumer side must be as efficient as possible. When source code and the source-level analysis are available, we could imagine utilizing this optimization in our framework as well.

6 Conclusion and Future Work

We have described a flexible and modular methodology for building assembly code analyses based on a novel notion of cooperating decompilers. We have shown the effectiveness of our framework through three example decompiler pipelines that share low-level components: for the output of `gcc`, `gcj`, and compilers for the Cool object-oriented language.

Primarily for program understanding, one might consider building a structural analysis decompiler module on top of the existing pipelines that could recover typical source-level control-flow constructs. However, a loop analysis decompiler that leaves the control-flow unstructured may also be useful for higher-level analyses.

We are particularly interested in assembly-level analyses for addressing mobile-code safety [Nec97,MWCG99], ideally in a foundational but also practical manner. As such, we have also designed our decompilation framework with soundness in mind (e.g., making decompilers work one instruction at a time and working in the framework of abstract interpretation). To achieve this, we envision building on our prior work on *certified program analyses* [CCN06], as well as drawing on abstract interpretation-based transformations [CC02,Riv03]. Such a modularization of code as we have achieved will likely be critical for feasibly proving the soundness of analysis implementations in a machine-checkable manner. This motivation also partly justifies our use of reflection tables produced by `gcj` or debugging information from `gcc`, as it seems reasonable to trade-off, at least, some annotations for safety checking.

Decompilation even beyond C or Java-like source code may also serve as a convenient methodology for structuring program analyses and verifiers. For example, one might imagine decompiler modules that translate certain mutable data structures into functional ones or uses of locks into atomic sections.

References

Aik96. Alexander Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–24, July 1996.

App01. Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

AWZ88. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–11, 1988.

AZ05. Paul Anderson and Mark Zarins. The CodeSurfer software understanding platform. In *International Workshop on Program Comprehension (IWPC)*, pages 147–148, 2005.

BL05. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 82–87, 2005.

BR04. Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In *Conference on Compiler Construction (CC)*, pages 5–23, 2004.

BRK+05. Gogul Balakrishnan, Thomas W. Reps, Nick Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, Suan Hsi Yong, Chi-Hua Chen, and Tim Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Conference on Computer-Aided Verification (CAV)*, pages 158–163, 2005.

CC77. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 234–252, 1977.

CC79. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

CC02. Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Symposium on Principles of Programming Languages (POPL)*, pages 178–190, 2002.

CCH94. Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. Combinations of abstract domains for logic programming. In *Symposium on Principles of Programming Languages (POPL)*, pages 227–239, 1994.

CCN06. Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 174–189, 2006.

CCNS05. Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. Type-based verification of assembly language for compiler debugging. In *Workshop on Types in Language Design and Implementation (TLDI)*, pages 91–102, 2005.

CL05. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 147–163, 2005.

CLN+00. Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 95–107, 2000.

CMB+95. Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria J. García de la Banda, and Manuel V. Hermenegildo. Improving abstract interpretations by combining domains. *ACM Trans. Program. Lang. Syst.*, 17(1):28–44, 1995.

CSF98. Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *International Conference on Software Maintenance (ICSM)*, pages 228–237, 1998.

FTA02. Jeffrey Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2002.

GN04. Sumit Gulwani and George C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis Symposium (SAS)*, pages 212–227, 2004.

GTN04. Sumit Gulwani, Ashish Tiwari, and George C. Necula. Join algorithms for the theory of uninterpreted functions. In *Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 311–323, 2004.

HJM+02. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Conference on Computer-Aided Verification (CAV)*, pages 526–538, 2002.

IDA. IDA Pro disassembler. http://www.datarescue.com/idabase.

JW04. Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134, 2004.

LGC02. Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Symposium on Principles of Programming Languages (POPL)*, pages 270–282, 2002.

LST02. Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Trans. Program. Lang. Syst.*, 24(2):112–152, 2002.

LY97. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

Mil78. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

MWCG99. J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

Myc99. Alan Mycroft. Type-based decompilation. In *European Symposium on Programming (ESOP)*, pages 208–223, 1999.

Nec97. George C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.

NO79. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

Riv03. Xavier Rival. Abstract interpretation-based certification of assembly code. In *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 41–55, 2003.

TC02. Jens Tröger and Cristina Cifuentes. Analysis of virtual method invocation for binary translation. In *Working Conference on Reverse Engineering (WCRE)*, pages 65–74, 2002.

VRCG+99. Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, page 13, 1999.

A The Product Decompiler

To clarify how the decompiler modules interact to advance simultaneously, we sketch here the product decompiler that ties together the pipeline. Let $\mathcal{L}$ be a decompiler that translates assembly instructions $I_{\mathcal{C}}$ to instructions in an intermediate language $I_{\mathcal{L}}$, and let $\mathcal{H}$ be a higher-level decompiler that translates $I_{\mathcal{L}}$ instructions into a higher-level IL instructions $I_{\mathcal{H}}$. The product decompiler is then a decompiler from the assembly language $I_{\mathcal{C}}$ to the higher-level IL $I_{\mathcal{H}}$. To indicate a sequence of instructions, we write I^* and use $\texttt{step}^* : \texttt{curr} \times (\texttt{instr}_{in}\ \texttt{list}) \to \texttt{abs}$ as a lifting of `step` that gives the abstract transition relation for straight-line code (i.e., a sequence of without control-flow instructions). Also, for presentation purposes, we consider type `succ` to be a finite map from locations to pairs of an abstract-state and a reinterpretation (with such a mapping written as $\ell \mapsto (a, \mathit{reinterp})$) and abuse notation slightly by identifying lists with sets.

In order to construct query objects, every decompiler must also define a function $\texttt{getHints} : \texttt{hints}_{out} \times \texttt{abs} \to \texttt{hints}_{in}$ that returns the `hints` object described in Sec. 2 for its input language. Observe that a decompiler produces the $\texttt{hints}_{in}$ object (for its input language) by either consulting its abstract state or by forwarding queries to higher-level decompilers (through the given $\texttt{hints}_{out}$).

The abstract state of the product decompiler is simply the pair of abstract states of the sub-decompilers. For `step`, we first call $\texttt{getHints}_{\mathcal{H}}$ to get this callback object for $\mathcal{L}$ (line 1). We then make a transition in $\mathcal{L}$ to generate the intermediate instruction $I_{\mathcal{L}}$ in order to make a transition in $\mathcal{H}$ (lines 2 and 3). Note that our actual implementation allows one-to-many decompilations (many-to-one can be obtained by using one-to-none). The output instruction for the product decompiler is simply from the output of $\texttt{step}_{\mathcal{H}}$ (line 4), while the *combine* collects together the successors. If $\texttt{step}_{\mathcal{H}}$ yielded a reinterpretation at ℓ, then we get the successor state for $\mathcal{L}$ by re-running $\texttt{step}_{\mathcal{L}}$ with the reinterpretation instructions I^*_C (line 10); otherwise, ℓ should be in Lo'.

type $\texttt{abs} = \texttt{abs}_{\mathcal{L}} \times \texttt{abs}_{\mathcal{H}}$

fun `step` $((q_{\mathcal{H}}, (lo, hi)), I_C) =$

1 **let** $q_{\mathcal{L}} = \texttt{getHints}_{\mathcal{H}}(q_{\mathcal{H}}, hi)$ **in**
2 **let** $I_{\mathcal{L}}, Lo' = \texttt{step}_{\mathcal{L}}((q_{\mathcal{L}}, lo), I_C)$ **in**
3 **let** $I_{\mathcal{H}}, Hi' = \texttt{step}_{\mathcal{H}}((q_{\mathcal{H}}, hi), I_{\mathcal{L}})$ **in**
4 $I_{\mathcal{H}}, \mathit{combine}(Lo' \cup \mathit{reinterp}(Hi'), Hi')$
5 where $\mathit{combine}(Lo', Hi') =$
6 $[\ \ell \mapsto (lo', hi')\ \mid$
7 $lo' = Lo'(\ell)\ \wedge\ hi' = Hi'(\ell)\ ]$
8 and $\mathit{reinterp}(Hi') =$
9 $[\ \ell \mapsto lo'\ \mid\ Hi'(\ell) = (\_, \texttt{Some}(I^*_C))$
10 $\wedge\ lo' = \texttt{step}^*_{\mathcal{L}}((q_{\mathcal{L}}, lo), I^*_C)\ ]$