A Typed Interrupt Calculus

Jens Palsberg Di Ma

Department of Computer Science
Purdue University, W. Lafayette, IN 47907
{palsberg,madi}@cs.purdue.edu

Abstract. Most real-time systems require responsive interrupt handling. Programming of interrupt handlers is challenging: in order to ensure responsiveness, it is often necessary to have interrupt processing enabled in the body of lower priority handlers. It would be a programming error to allow the interrupt handlers to interrupt each other in a cyclic fashion; it could lead to an unbounded stack. Until now, static checking for such errors could only be done using model checking. However, the needed form of model checking requires a whole-program analysis that cannot check program fragments. In this paper, we present a calculus that contains essential constructs for programming interrupt-driven systems. The calculus has a static type system that guarantees stack boundedness and enables modular type checking. A number of common programming idioms have been type checked using our prototype implementation.

1 Introduction

Interrupt-driven systems Interrupts and interrupt handlers are often used in systems where fast response to an event is essential. Embedded interrupt-driven systems generally have a fixed number of interrupt sources with a handler defined for each source. When an interrupt occurs, control is transferred automatically to the handler for that interrupt source, unless interrupt processing is disabled. If disabled, the interrupt will wait for interrupt processing to be enabled. To get fast response times, it is necessary to keep interrupt processing enabled most of the time, including in *the body* of lower priority interrupt handlers. This opens the door for interrupt handlers to be themselves interrupted, making it difficult to understand whether real-time deadlines can be met. Conversely, to write reliable code with a given real-time property, it is simplest to disable interrupts in the body of interrupt handlers. This may delay the handling of other interrupts and therefore make it difficult for the system to have other desired real-time properties. The resultant tension between fast response times and easy-to-understand, reliable code drives developers to write code that is often difficult to test and debug.

A nasty programming error A particularly nasty programming error is to allow the interrupt handlers to interrupt each other indefinitely, leading to an unbounded stack. For example, consider the following two interrupt handlers.

```
handler 1 {                          handler 2 {
  // do something                      // do something
  enable-handling-of-interrupt-2       enable-handling-of-interrupt-1
  // do something else                 // do something else
  iret                                 iret
}                                    }
```

Suppose an interrupt from source 1 arrives first, and so handler 1 is called. Before returning, handler 1 enables handling of interrupts from source 2, and unfortunately an interrupt from source 2 arrives before handler 1 has returned. Thus handler 2 is called, and it, in turn, enables handling of interrupts from source 1 before returning, allowing a highly undesirable cycle and an unbounded stack.

A traditional type system does not check for this kind of error. The error is not about misusing data; it is about needing unbounded resources. Until now, static checking for such errors could only be done using model checking. However, the needed form of model checking is a whole-program analysis that cannot check program fragments [13, 3]. The goal of this paper is to present a type system that guarantees stack boundedness and enables modular type checking. To do that, we need a minimal setting in which to study interrupt-driven systems.

The need for a new calculus For many programming paradigms, there is a small calculus which allows the study of properties in a language-independent way and which makes it tractable to prove key properties. For example, for functional programming there is the λ-calculus [2], for concurrent programming there is Milner's calculus of communicating systems [7], for object-oriented programming there is the Abadi-Cardelli object calculus [1], and for mobile computation there is the π-calculus [8] and the ambient calculus [4]. These calculi do not offer any notion of interrupts and interrupt handling. While such concepts might be introduced on top of one of those calculi, we believe that it is better to design a new calculus with interrupts at the core. A new calculus should focus on the essential concepts and leave out everything else.

What are the essential concepts? While some modern, general-purpose CPUs have sophisticated ways of handling internal and external interrupts, the notion of an interrupt mask register (imr) is widely used. This is especially true for CPUs that are used in embedded systems with small memory size, a need for low power consumption, and other constraints. The following table lists some characteristics of four CPUs that are often used in embedded systems:

Well-known product	Processor	# of interrupt sources	master bit
Microcontroller	Zilog Z86	6	yes
iPAQ Pocket PC	Intel strongARM, XScale	21	no
Palm	Motorola Dragonball (68000 Family)	22	yes
Microcontroller	Intel MCS-51 Family (8051 etc)	6	yes

Each of these processors have similar-looking imr's. For example, consider the imr for the MCS-51 (which calls it the interrupt enable (IE) register):

EA	–	ET2	ES	ET1	EX1	ET0	EX0

The bits have the following meanings:

- EA: enable/disable all interrupt handling,
- –: reserved (not used), and
- each of the remaining six bits corresponds to a specific interrupt source.

We will refer to the EA bit (and similar bits on other processors) as the *master bit*. The idea is that for a particular interrupt handler to be enabled, *both* the master bit and the bit for that interrupt handler have to be enabled. This semantics is supported by the Z86, Dragonball, MCS-51, and many other processors. When an interrupt handler is called, a return address is stored on the stack, and the processor automatically turns *off* the master bit. At the time of return, the processor turns the master bit back *on*. Not all processors use this scheme: the strongARM does not have a master bit. In this paper we focus on modeling processors that do have a master bit.

In the rest of the paper, we will use a representation of the imr which is independent of particular architectures. We represent the imr as a bit sequence $b_0 b_1 \ldots b_n$, where $b_i \in \{0, 1\}$, b_0 is the master bit, and, for $i > 0$, b_i is the bit for interrupts from source i which is handled by handler i. Notice that the master bit is the most significant bit, and that the bit for handler 1 is the second-most significant bit, and so on. This layout is different from some processors, and it simplifies the notation used later.

Our Results We present a calculus that contains essential constructs for programming interrupt-driven systems. A program in the calculus consists of a main part and some interrupt handlers. A program execution has access to:

- an interrupt mask register that can be updated during computation,
- a stack for storing return addresses, and
- a memory of integer variables; output is done via memory-mapped I/O.

The calculus is intended for modeling embedded systems that should run "forever," and for which termination would be considered a disastrous error. To model that, the calculus is designed such that no program can terminate; nontermination is guaranteed.

Each element on the stack is a return address. When we measure the size of the stack, we simply count the number of elements on the stack.

For our calculus, we present a type system that guarantees stack boundedness and enables modular type checking. A number of common programming idioms have been type checked using our prototype implementation.

A type for a handler contains information about the imr on entry and the imr at the point of return. Given that a handler can be called at different points

in the program where the imr may have different values, the type of a handler is an intersection type [5, 6] of the form:

$$\bigwedge_{j=1}^{n}((\widehat{imr})^j \xrightarrow{\delta^j} (\widehat{imr}')^j).$$

where the j^{th} component of the intersection means:

> if the handler is called in a situation where the imr can be conservatively approximated by $(\widehat{imr})^j$, then at the point of return, the imr can conservatively be approximated by $(\widehat{imr}')^j$, and during that call, the stack will grow by at most δ^j elements, excluding the return address for the call itself.

The annotations δ^j help with checking that the stack is bounded. Our type system with annotated types is an example of a type-based analysis [10].

Rest of the paper In Section 2 we introduce our interrupt calculus and type system via six examples. In Section 3 we present the syntax and semantics of the interrupt calculus, and we prove that no program can terminate. In Section 4 we present our type system and prove stack boundedness, and in Section 5 we conclude. In the appendix we prove the theorem of type preservation which is a key lemma in the proof of stack boundedness.

2 Examples

We will introduce our interrupt calculus and type system via six examples of increasing sophistication. The first five programs have been type checked using our prototype implementation, while the sixth program illustrates the limitations of our type system. We will use the concrete syntax that is supported by our type checker; later, in Sections 3–4, we will use an abstract syntax. Note that an imr value, say, 11 will in the concrete syntax be written as `11b`, to remind the reader that it is a binary value. Also, the type of a handler

$$\bigwedge_{j=1}^{n}((\widehat{imr})^j \xrightarrow{\delta^j} (\widehat{imr}')^j).$$

will be written `(`$(\widehat{imr})^1$ `->` $(\widehat{imr}')^1$ `:` δ^1`)` `...` `(`$(\widehat{imr})^n$ `->` $(\widehat{imr}')^n$ `:` δ^n`).`

The program in Figure 1 is a version of Example 3-5 from Wayne Wolf's textbook [14, p.113]. The program copies data from one device to another device. The program uses memory-mapped I/O; two variables map to the device registers:

- `indata`: the input device writes data in this register and
- `outdata`: the output device reads data from this register.

```
Maximum stack size: 1

imr = imr or 11b
loop {                                  handler 1 [ ( 11b -> 11b : 0 ) ] {
   if ( gotchar == 0 ) {                   achar    = indata
        outdata   = achar                  gotchar  = 0
        gotchar   = 1                      iret
   } else {                             }
        skip
   }
}
```

Fig. 1. A program for copying data from one device to another device.

```
maximum stack size: 1                   handler 1 [ ( 111b -> 111b : 0 ) ] {
                                           skip
imr = imr or 111b                          iret
loop {                                  }
   skip                                 handler 2 [ ( 111b -> 111b : 0 ) ] {
   imr = imr or 111b                       skip
}                                          iret
                                        }
```

Fig. 2. Two selfish handlers

The line `maximum stack size: 1` is a part of the program text. It tells the type checker to check that the stack can never be of a size greater than one. The number 1 is a count of return addresses on the stack; nothing else than return addresses can be put on the stack in our calculus. The header of the handler contains the annotation `11b -> 11b : 0`. It is a type that says that if the handler is called in a situation where the imr can be conservatively approximated by 11, then it will return in a situation where the imr can be conservatively approximated by 11, and the stack will not grow during the call. The value 11 should be read as follows. The leftmost bit is the master bit, and the next bit is the bit for handler 1. The value 11 means that handler 1 is enabled.

The program in Figure 2 is much like the program in Figure 1, except that there are now two handlers. The handlers cannot be interrupted so the maximum stack size is 1. Notice that since there are two handlers, the imr has three bits. They are organized as follows. The leftmost bit is, as always, the master bit. The next bit is the bit for handler 1, and the rightmost bit is the bit for handler 2.

The program in Figure 3 illustrates how to program a notion of prioritized handlers where handler 1 is of higher priority than handler 2. While handler 1 cannot be interrupted by handler 2, it is possible for handler 2 to be interrupted by handler 1. Handler 2 achieves that by disabling its own bit in the imr with the statement `imr = imr and 110b`, and then enabling the master bit with the statement `imr = imr or 100b`. Thus, handler 2 can be interrupted before it

```
                                    handler 1 [ ( 111b -> 111b : 0 )
                                                ( 110b -> 110b : 0 ) ] {
maximum stack size: 2                  skip
                                       iret
imr = imr or 111b                   }
loop {                              handler 2 [ ( 111b -> 111b : 1 ) ] {
   skip                                skip
   imr = imr or 111b                   imr = imr and 110b
}                                      imr = imr or  100b
                                       iret
                                    }
```

Fig. 3. Two prioritized handlers

```
                                    handler 1 [ ( 111b -> 101b : 1 )
                                                ( 110b -> 100b : 0 ) ] {
maximum stack size: 2                  imr = imr and 101b
                                       imr = imr or  100b
imr = imr or 111b                      iret
loop {                              }
   imr = imr or 111b                handler 2 [ ( 111b -> 110b : 1 )
}                                               ( 101b -> 100b : 0 ) ] {
                                       imr = imr and 110b
                                       imr = imr or  100b
                                       iret
                                    }
```

Fig. 4. Two cooperative handlers

returns. Accordingly, the maximum stack size is 2. The type for handler 1 is an intersection type that reflects that handler 1 can be called both from the main part and from handler 2. If it is called from the main part, then the imr is 111, and if it is called from handler 2, then the imr is 110. The type for handler 2 has annotation 1 because handler 2 can be interrupted by handler 1, which, in turn, cannot be interrupted.

The program in Figure 4 illustrates how both handlers can allow the other handler to interrupt. Each handler uses the discipline of disabling its own bit in the imr before setting the master bit to 1. This ensures that the maximum stack size is two.

The program in Figure 5 illustrates that n handlers can lead to a bounded stack where the bound is greater than n. In this case we have two handlers and a maximum stack size of three. A stack size of three is achieved by first calling handler 1, then calling handler 2, and finally calling handler 1 again.

While our type system can type check many common programming idioms, as illustrated above, there are useful programs that it cannot type check. For example, the program in Figure 6, written by Dennis Brylow, is a 60 second

```
maximum stack size: 3

imr = imr or 111b
loop {
   imr = imr or 111b
}
```

```
handler 1 [ ( 111b -> 111b : 2 )
            ( 110b -> 100b : 0 ) ] {
   imr = imr and 101b
   imr = imr or  100b
   iret
}
handler 2 [ ( 111b -> 100b : 1 )
            ( 101b -> 100b : 1 ) ] {
   imr = imr and 110b
   imr = imr or  010b
   imr = imr or  100b
   imr = imr and 101b
   iret
}
```

Fig. 5. Two fancy handlers

```
maximum stack size: 1

SEC = SEC + 60
imr = imr or 110b
loop {
   if( SEC == 0 ) {
   OUT = 1
   imr = imr and 101b
   imr = imr or  001b
   } else {
   OUT = 0
   }
}
```

```
handler 1 [ ( 111b -> 111b : 0 )
            ( 110b -> 110b : 0 ) ] {
   SEC = SEC +  (-1)
   iret
}
handler 2 [ ( 111b -> 110b : 0 )
            ( 101b -> 110b : 0 ) ] {
   SEC = 60
   imr = imr and 110b
   imr = imr or  010b
   iret
}
```

Fig. 6. A timer

timer. The `OUT` variable will be 0 for 60 seconds after a request for interrupt 2. There are two interrupt handlers:

- The first handler is for an external timer that is expected to request an interrupt once each second.
- The second handler is a trigger. When it arrives, the `OUT` variable will become 0 for 60 seconds. Then `OUT` will become 1, and remain so until the next trigger event.

Our type system cannot handle this pattern where handler 2 disables itself and then enables handler 1, and where the main program disables handler 1 and enables handler 2. Thus, while the program in Figure 6 has a maximum stack size of 2, it does not type check in our type system.

3 The Interrupt Calculus

3.1 Syntax

We will use an abstract syntax that is slightly more compact than the concrete syntax used in Section 2.

We use x to range over a set of program variables, we use imr to range over bit strings, and we use c to range over integer constants.

$$\begin{array}{lll}
\text{(program)} & p & ::= (m, \overline{h}) \\
\text{(main)} & m & ::= \mathsf{loop}\ s \mid s\ ;\ m \\
\text{(handler)} & h & ::= \mathsf{iret} \mid s\ ;\ h \\
\text{(statements)} & s & ::= x = e \mid \mathsf{imr} = \mathsf{imr} \wedge imr \mid \mathsf{imr} = \mathsf{imr} \vee imr \mid \\
 & & \quad \mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2 \mid s_1\ ;\ s_2 \mid \mathsf{skip} \\
\text{(expression)} & e & ::= c \mid x \mid x + c \mid x_1 + x_2
\end{array}$$

The over bar notation $\overline{h}$ denotes a sequence $h_1 \ldots h_n$; we will use the notation $\overline{h}(i) = h_i$.

We use a to range over m and h. We identify programs that are equivalent under the smallest congruence generated by the rules:

$$\begin{array}{c}
(s_1\ ;\ s_2)\ ;\ m = s_1\ ;\ (s_2\ ;\ m) \\
(s_1\ ;\ s_2)\ ;\ h = s_1\ ;\ (s_2\ ;\ h) \\
(s_1\ ;\ s_2)\ ;\ s = s_1\ ;\ (s_2\ ;\ s).
\end{array}$$

With these rules, we can rearrange any m or h into one of the seven forms:

$$\mathsf{loop}\ s \qquad \mathsf{iret} \qquad x = e; a \qquad \mathsf{imr} = \mathsf{imr} \wedge imr; a \qquad \mathsf{imr} = \mathsf{imr} \vee imr; a$$
$$(\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2); a \qquad \mathsf{skip}; a.$$

3.2 Semantics

We use R to denote a *store*, that is, a partial function mapping program variables to integers.

We use σ to denote a *stack* generated by the grammar: $\sigma ::= \mathsf{nil} \mid a :: \sigma$. We define the size of a stack as follows: $|\mathsf{nil}| = 0$ and $|a :: \sigma| = 1 + |\sigma|$.

If $imr = b_0 b_1 \ldots b_n$, where $b_i \in \{0, 1\}$, then we will use the notation $imr(i) = b_i$. The predicate *enabled* is defined as follows:

$$enabled(imr, i) = (imr(0) = 1) \wedge (imr(i) = 1), \qquad i \in 1..n.$$

We use 0 to denote the imr value where all bits are 0. We use t_i to denote the imr value where all bits are 0's except that the ith bit is set to 1. We will use $\wedge$ to denote bitwise logical conjunction, $\vee$ to denote bitwise logical disjunction, $\leq$ to denote bitwise logical implication, and $(\cdot)^\bullet$ to denote bitwise logical negation. Notice that $enabled(\mathsf{t}_0 \vee \mathsf{t}_i, j)$ is true for $i = j$ and false otherwise. The imr values, ordered by $\leq$, form a lattice with bottom element 0.

A *program state* is a tuple $\langle \overline{h}, R, imr, \sigma, a\rangle$. We will refer to a as *the current statement*; it models the instruction pointer of a CPU. We use P to range over program states. If $P = \langle \overline{h}, R, imr, \sigma, a\rangle$, then we use the notation $P.stk = \sigma$. For $p = (m, \overline{h})$, the initial program state for executing p is $P_p = \langle \overline{h}, \lambda x.0, 0, \mathsf{nil}, m\rangle$, where the function $\lambda x.0$ is defined on the variables that are used in the program p.

A small-step operational semantics for the language is given by the reflexive, transitive closure of the relation $\rightarrow$ on program states:

$$\langle \overline{h}, R, imr, \sigma, a\rangle \rightarrow \langle \overline{h}, R, imr \wedge \mathsf{t}_0^\bullet, a :: \sigma, \overline{h}(i)\rangle \quad \text{if } \mathit{enabled}(imr, i) \tag{1}$$

$$\langle \overline{h}, R, imr, \sigma, \mathsf{iret}\rangle \rightarrow \langle \overline{h}, R, imr \vee \mathsf{t}_0, \sigma', a\rangle \quad \text{if } \sigma = a :: \sigma' \tag{2}$$

$$\langle \overline{h}, R, imr, \sigma, \mathsf{loop}\ s\rangle \rightarrow \langle \overline{h}, R, imr, \sigma, s; \mathsf{loop}\ s\rangle \tag{3}$$

$$\langle \overline{h}, R, imr, \sigma, x = e; a\rangle \rightarrow \langle \overline{h}, R\{x \mapsto \mathit{eval}_R(e)\}, imr, \sigma, a\rangle \tag{4}$$

$$\langle \overline{h}, R, imr, \sigma, \mathsf{imr} = \mathsf{imr} \wedge imr'; a\rangle \rightarrow \langle \overline{h}, R, imr \wedge imr', \sigma, a\rangle \tag{5}$$

$$\langle \overline{h}, R, imr, \sigma, \mathsf{imr} = \mathsf{imr} \vee imr'; a\rangle \rightarrow \langle \overline{h}, R, imr \vee imr', \sigma, a\rangle \tag{6}$$

$$\langle \overline{h}, R, imr, \sigma, (\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2); a\rangle \rightarrow \langle \overline{h}, R, imr, \sigma, s_1; a\rangle \quad \text{if } R(x) = 0 \tag{7}$$

$$\langle \overline{h}, R, imr, \sigma, (\mathsf{if0}\ (x)\ s_1\ \mathsf{else}\ s_2); a\rangle \rightarrow \langle \overline{h}, R, imr, \sigma, s_2; a\rangle \quad \text{if } R(x) \neq 0 \tag{8}$$

$$\langle \overline{h}, R, imr, \sigma, \mathsf{skip}; a\rangle \rightarrow \langle \overline{h}, R, imr, \sigma, a\rangle \tag{9}$$

We define the function $\mathit{eval}_R(e)$ as follows:

$$\begin{aligned} \mathit{eval}_R(c) &= c \\ \mathit{eval}_R(x) &= R(x) \\ \mathit{eval}_R(x + c) &= R(x) + c \\ \mathit{eval}_R(x_1 + x_2) &= R(x_1) + R(x_2). \end{aligned}$$

Rule (1) models that if an interrupt is enabled, then it may occur. The rule says that if $\mathit{enabled}(imr, i)$, then it is a possible transition to push the current statement on the stack, make $\overline{h}(i)$ the current statement, and turn off the master bit in the imr. Notice that we make no assumptions about the interrupts arrivals; any enabled interrupt can occur at any time, and, conversely, no interrupt has to occur.

Rule (2) models interrupt return. The rule says that to return from an interrupt, remove the top element of the stack, make the removed top element the current statement, and turn on the master bit.

Rule (3) is an unfolding rule for loops, and Rules (4)–(9) are standard rules for statements.

3.3 Nontermination

We say that a program p *can terminate* if $P_p \rightarrow^* P'$ and there is no P'' such that $P' \rightarrow P''$.

We say that a program state $\langle \overline{h}, R, imr, \sigma, a\rangle$ is *consistent* if and only if (1) $\sigma = \mathsf{nil}$ and $a = m$; or (2) $\sigma = h^k :: \ldots :: h^1 :: m :: \mathsf{nil}$ and $a = h$, for $k \geq 0$, where $k = 0$ means $\sigma = m :: \mathsf{nil}$.

Lemma 1. (Consistency Preservation) *If P is consistent and $P \rightarrow P'$, then P' is consistent.*

Proof. A straightforward cases analysis of $P \rightarrow P'$.

Lemma 2. (Progress) *If P is consistent, then there exists P' such that $P \rightarrow P'$.*

Proof. There are two cases of P:

- $P = \langle \overline{h}, R, imr, \mathsf{nil}, m\rangle$. There are two cases of m:
  - if $m = \mathsf{loop}\ s$, then Rule (3) gives $P' = \langle \overline{h}, R, imr, \mathsf{nil}, s; \mathsf{loop}\ s\rangle$, and
  - if $m = s; m'$, then Rules (4)–(9) ensure that there exists a state P' such that $P \rightarrow P'$.
- $P = \langle \overline{h}, R, imr, h^k :: \ldots :: h^1 :: m :: \mathsf{nil}, h\rangle$, $k \geq 0$. There are two cases of h:
  - if $h = \mathsf{iret}$, then either $k = 0$ and $s = m :: \mathsf{nil}$, and Rule (2) gives $P' = \langle \overline{h}, R, imr \vee \mathsf{t}_0, \mathsf{nil}, m\rangle$, or $k > 0$ and hence $P' = \langle \overline{h}, R, imr \vee \mathsf{t}_0, h^{k-1} :: \ldots :: h^1 :: m :: \mathsf{nil}, h^k\rangle$, and
  - if $h = s; h'$, then Rules (4)–(9) ensure that there exists a state P' such that $P \rightarrow P'$.

Theorem 1. (Nontermination) *No program can terminate.*

Proof. Suppose a program p can terminate, that is, suppose $P_p \rightarrow^* P'$ and there is no P'' such that $P' \rightarrow P''$. Notice first that P_p is consistent by consistency criterion (1). From Lemma 1 and induction on the number of execution steps in $P_p \rightarrow^* P'$, we have that P' is consistent. From Lemma 2 we have that there exists P'' such that $P' \rightarrow P''$, a contradiction.

4 Type System

4.1 Types

We will use imr values as types. When we intend an imr value to be used as a type, we will use the mnemonic device of writing it with a hat, for example, $\widehat{imr}$.

We will use the bitwise logical implication $\leq$ as the subtype relation. For example, $101 \leq 111$. We will also use $\leq$ to specify the relationship between an imr value and its type. When we want to express that an imr value imr has type $\widehat{imr}$, we will write $imr \leq \widehat{imr}$. The meaning is that $\widehat{imr}$ is a conservative approximation of imr, that is, if a bit in imr is 1, then the corresponding bit in $\widehat{imr}$ is also 1.

We use K to range over the integers, and we use δ to range over the nonnegative integers.

We use τ to range over intersection types of the form:

$$\bigwedge_{j=1}^{n}((\widehat{imr})^j \xrightarrow{\delta^j} (\widehat{imr}')^j).$$

We use $\overline{\tau}$ to range over a sequence $\tau_1 \ldots \tau_n$; we will use the notation $\overline{\tau}(i) = \tau_i$.

4.2 Type Rules

We will use the following forms of type judgments:

Type Judgment	Meaning
$\overline{\tau} \vdash h \ : \ \tau$	Interrupt handler h has type τ
$\overline{\tau}, \widehat{imr} \ \vdash_K \ \sigma$	Stack σ type checks
$\overline{\tau}, \widehat{imr} \ \vdash_K \ m$	Main part m type checks
$\overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}'$	Handler h type checks
$\overline{\tau}, \widehat{imr} \ \vdash_K \ s \ : \ \widehat{imr}'$	Statement s type checks
$\overline{\tau} \ \vdash_K \ P$	Program state P type checks

The judgment $\overline{\tau}, \widehat{imr} \ \vdash_K \ m$ means that if the handlers are of type $\overline{\tau}$, and the imr has type $\widehat{imr}$, then m type checks. The integer K bounds the stack size to be at most K. We can view K as a "stack budget" in the sense that any time an element is placed on the stack, the budget goes down by one, and when an element is removed from the stack, the budget goes up by one. The type system ensures that the budget does not go below zero.

The judgment $\overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}'$ means that if the handlers are of type $\overline{\tau}$, and the imr has type $\widehat{imr}$, then h type checks, and at the point of returning from the handler, the imr has type $\widehat{imr}'$. The integer K means that during the call, the stack will grow by at most K elements. Notice that "during the call" may include calls to other interrupt handlers.

The judgment $\overline{\tau}, \widehat{imr} \ \vdash_K \ s \ : \ \widehat{imr}'$ has a meaning similar to that of $\overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}'$.

A judgment for a program state is related to the concrete syntax used in Section 2 in the following way. We can dissect the concrete syntax into four parts: (1) a maximum stack size K, (2) the types $\overline{\tau}$ for the handlers, (3) a main part m, and (4) a collection $\overline{h}$ of handlers. When we talk about a program $(m, \overline{h})$ in the abstract syntax, the two other parts K and $\overline{\tau}$ seem left out. However, they reappear in the judgment: $\overline{\tau} \ \vdash_K \ P_{(m,\overline{h})}$. Thus, that judgment can be read simply as: "the program type checks."

For two sequences $\overline{h}$, $\overline{\tau}$ of the same length, we will use the abbreviation:

$$\overline{\tau} \vdash \overline{h} \ : \ \overline{\tau}$$

to denote the family of judgments

$$\overline{\tau} \vdash \overline{h}(i) \ : \ \overline{\tau}(i)$$

for all i in the common domain of $\overline{h}$ and $\overline{\tau}$.

We will use the abbreviation:

$$safe(\overline{\tau}, \widehat{imr}, K) = \begin{bmatrix} \forall i \in 1 \ldots n \\ \mathsf{if}\ enabled(\widehat{imr}, i) \\ \mathsf{then,\ whenever}\ \overline{\tau}(i) = \ldots \bigwedge (\widehat{imr} \xrightarrow{\delta} \widehat{imr}')\ \bigwedge \ldots, \\ \mathsf{we\ have}\ \widehat{imr}' \leq \widehat{imr}\ \wedge\ \delta + 1 \leq K \end{bmatrix}.$$

The idea of $safe(\overline{\tau}, \widehat{imr}, K)$ is to guarantee that it is safe for an interrupt to occur. If an interrupt occurs at a time when the imr has type $\widehat{imr}$ and the "stack budget" is K, then the handler for that interrupt should return with an imr that has a type which is a *subtype* of $\widehat{imr}$. This ensures that $\widehat{imr}$ is still a type for the imr after the interrupt. Moreover, the stack should grow at most δ elements during the call, plus a return address for the call itself.

As a mnemonic, we will sometimes use imr_r for the return imr value of an interrupt handler, and we will sometimes use imr_b for the imr value when an interrupt handler is called.

$$\frac{\overline{\tau}\ \vdash\ \overline{h}\ :\ \overline{\tau} \qquad imr \leq \widehat{imr} \qquad \overline{\tau}, \widehat{imr}\ \vdash_K\ m}{\overline{\tau}\ \vdash_K\ \langle \overline{h}, R, imr, \mathsf{nil}, m \rangle} \tag{10}$$

$$\frac{\begin{array}{c}\overline{\tau}\ \vdash\ \overline{h}\ :\ \overline{\tau} \quad imr \leq \widehat{imr} \\ \overline{\tau}, \widehat{imr}\ \vdash_K\ h\ :\ \widehat{imr}_r \quad \widehat{imr}_r \leq \widehat{imr}_b \quad \overline{\tau}, \widehat{imr}_b\ \vdash_K\ \sigma\end{array}}{\overline{\tau}\ \vdash_K\ \langle \overline{h}, R, imr, \sigma, h \rangle} \tag{11}$$

$$\frac{\overline{\tau}, \widehat{imr}\ \vdash_{K+1}\ m}{\overline{\tau}, \widehat{imr}\ \vdash_K\ m :: \mathsf{nil}} \tag{12}$$

$$\frac{\overline{\tau}, \widehat{imr}\ \vdash_{K+1}\ h\ :\ \widehat{imr}_r \qquad \widehat{imr}_r \leq \widehat{imr}_b \qquad \overline{\tau}, \widehat{imr}_b\ \vdash_{K+1}\ \sigma}{\overline{\tau}, \widehat{imr}\ \vdash_K\ h :: \sigma} \tag{13}$$

$$\frac{\overline{\tau}, (\widehat{imr})^j \wedge \mathsf{t}_0^\bullet\ \vdash_{\delta^j}\ h\ :\ (\widehat{imr}')^j \qquad j \in 1..n}{\overline{\tau}\ \vdash\ h :\ \bigwedge_{j=1}^{n} ((\widehat{imr})^j \xrightarrow{\delta^j} (\widehat{imr}')^j)} \tag{14}$$

$$\frac{\overline{\tau}, \widehat{imr}\ \vdash_K\ s\ :\ \widehat{imr}}{\overline{\tau}, \widehat{imr}\ \vdash_K\ \mathsf{loop}\ s} \quad \Big[safe(\overline{\tau}, \widehat{imr}, K)\Big] \tag{15}$$

$$\frac{\overline{\tau}, \widehat{imr}\ \vdash_K\ s\ :\ \widehat{imr}' \qquad \overline{\tau}, \widehat{imr}'\ \vdash_K\ m}{\overline{\tau}, \widehat{imr}\ \vdash_K\ s; m} \tag{16}$$

$$\overline{\tau}, \widehat{imr}\ \vdash_K\ \mathsf{iret}\ :\ \widehat{imr} \vee \mathsf{t}_0 \quad \Big[safe(\overline{\tau}, \widehat{imr}, K)\Big] \tag{17}$$

$$\frac{\overline{\tau}, \widehat{imr}\ \vdash_K\ s\ :\ \widehat{imr}' \qquad \overline{\tau}, \widehat{imr}'\ \vdash_K\ h\ :\ \widehat{imr}''}{\overline{\tau}, \widehat{imr}\ \vdash_K\ s; h\ :\ \widehat{imr}''} \tag{18}$$

$$\overline{\tau}, \widehat{imr} \ \vdash_K \ x = e \ : \ \widehat{imr} \quad \Big[safe(\overline{\tau}, \widehat{imr}, K)\Big] \tag{19}$$

$$\overline{\tau}, \widehat{imr} \ \vdash_K \ \mathsf{imr} = \mathsf{imr} \wedge imr' \ : \ \widehat{imr} \wedge imr' \quad \Big[safe(\overline{\tau}, \widehat{imr}, K)\Big] \tag{20}$$

$$\overline{\tau}, \widehat{imr} \ \vdash_K \ \mathsf{imr} = \mathsf{imr} \vee imr' \ : \ \widehat{imr} \vee imr' \quad \Big[safe(\overline{\tau}, \widehat{imr}, K)\Big] \tag{21}$$

$$\frac{\overline{\tau}, \widehat{imr} \ \vdash_K \ s_1 \ : \ \widehat{imr}' \qquad \overline{\tau}, \widehat{imr} \ \vdash_K \ s_2 \ : \ \widehat{imr}'}{\overline{\tau}, \widehat{imr} \ \vdash_K \ \mathsf{if0} \ (x) \ s_1 \ \mathsf{else} \ s_2 \ : \ \widehat{imr}'} \quad \Big[safe(\overline{\tau}, \widehat{imr}, K)\Big] \tag{22}$$

$$\frac{\overline{\tau}, \widehat{imr} \ \vdash_K \ s_1 \ : \ \widehat{imr}_1 \qquad \overline{\tau}, \widehat{imr}_1 \ \vdash_K \ s_2 \ : \ \widehat{imr}_2}{\overline{\tau}, \widehat{imr} \ \vdash_K \ s_1; s_2 \ : \ \widehat{imr}_2} \tag{23}$$

$$\overline{\tau}, \widehat{imr} \ \vdash_K \ \mathsf{skip} \ : \ \widehat{imr} \quad \Big[safe(\overline{\tau}, \widehat{imr}, K)\Big] \tag{24}$$

Rules (10)–(11) are for type checking program states. The actual imr value imr is abstracted to a type $\widehat{imr}$ which is used to type check the current statement. In Rule (11), the last two hypotheses ensure that interrupts can return to their callers in a type-safe way. This involves type checking the stack, which is done by Rules (12)–(13).

Rule (14) says that the type of a handler is an intersection type so the handler must have all of the component types of the intersection. For each component type, the annotation δ^j is used as the bound on how much the stack can grow during a call to the handler. Notice that an intersection of different components cannot be reduced into a single component. The rule type checks the handler with the master bit initially turned off.

Rules (15)–(24) are type rules for statements. They are flow-sensitive to the imr, and most of them have the side condition $safe(\overline{\tau}, \widehat{imr}, K)$. The side condition ensures that if an enabled interrupt occurs, then the handler can both be called and return in a type-safe way.

4.3 Type Preservation and Stack Boundedness

Theorem 2. (Type Preservation) *Suppose P is a consistent program state. If $\overline{\tau} \vdash_K P$, $K \geq 0$, and $P \rightarrow P'$, then $\overline{\tau} \vdash_{K'} P'$ and $K' \geq 0$, where $K' = K + |P.stk| - |P'.stk|$.*

Proof. See Appendix A.

Theorem 3. (Multi-Step Type Preservation) *Suppose P is a consistent program state. If $\overline{\tau} \vdash_K P$, $K \geq 0$, and $P \rightarrow^* P'$, then $\overline{\tau} \vdash_{K'} P'$ and $K' \geq 0$, where $K' = K + |P.stk| - |P'.stk|$.*

Proof. We need to prove that

> $\forall n \geq 0$, if $\overline{\tau} \vdash_K P$, $K \geq 0$, and $P \rightarrow^n P'$, then $\overline{\tau} \vdash_{K'} P'$ and $K' \geq 0$, where $K' = K + |P.stk| - |P'.stk|$.

We proceed by induction on n. In the base case of $n = 0$, we have $P = P'$, so $K' = K + |P.stk| - |P.stk| = K$. From $P' = P$ and $K' = K$, we have $\overline{\tau} \vdash_{K'} P'$ and $K' \geq 0$.

In the induction step, assume that the property is true for n. Suppose $\overline{\tau} \vdash_K P$, $K \geq 0$, and $P \rightarrow^n P' \rightarrow P''$. From the induction hypothesis, we have $\overline{\tau} \vdash_{K'} P'$ and $K' \geq 0$, where

$$K' = K + |P.stk| - |P'.stk| \tag{25}$$

From Lemma 1 we have that P' is consistent. From Theorem 2, we have $\overline{\tau} \vdash_{K''} P''$ and $K'' \geq 0$, where

$$K'' = K' + |P'.stk| - |P''.stk| \tag{26}$$

From Equations (25) and (26), we have

$$\begin{aligned} K'' &= K' + |P'.stk| - |P''.stk| \\ &= K + |P.stk| - |P'.stk| + |P'.stk| - |P''.stk| \\ &= K + |P.stk| - |P''.stk| \end{aligned}$$

as desired.

Corollary 1. (Stack Boundedness) *If* $\overline{\tau} \vdash_K P_p$, $K \geq 0$, *and* $P_p \rightarrow^* P'$, *then* $|P'.stk| \leq K$.

Proof. Notice first that P_p is consistent. From $\overline{\tau} \vdash_K P_p$, $K \geq 0$, $P_p \rightarrow^* P'$, and Theorem 3, we have $\overline{\tau} \vdash_{K'} P'$ and $K' \geq 0$, where $K' = K + |P_p.stk| - |P'.stk|$. From $K' = K + |P_p.stk| - |P'.stk|$ and $|P_p.stk| = 0$, we have $K' = K - |P'.stk|$, so, since $K' \geq 0$, we have $|P'.stk| \leq K$, as desired.

5 Conclusion

Our calculus is a good foundation for studying interrupt-driven systems. In tune with the need of embedded systems, no program can terminate. Our type system guarantees stack boundedness, and a number of idioms have been type checked using our prototype implementation.

Our calculus can be viewed as the core of our ZIL language [9, 12]. ZIL is an intermediate language that strongly resembles Z86 assembly language except that it uses variables instead of registers. We use ZIL as an intermediate language in compilers.

Future work includes implementing the type checker for a full-scale language, such as ZIL, and experimenting with type checking production code. Another challenge is to design a more powerful type system which can type check the

timer program in Figure 6. It may be possible to integrate the interrupt calculus with a calculus such as the π-calculus. This could give the advantage that existing methods, techniques, and tools can be used.

To enable our type system to be used easily for legacy systems, we need a way to infer the types of all interrupt handlers. Such type inference may be related to model checking. An idea might be to first execute a variant of a model checking algorithm for pushdown automata [13, 3], and then use the computed information to construct types (for a possibly related result, see [11]). At present, it is open whether type inference for our type system is decidable.

Acknowledgment: Our work is supported by the National Science Foundation ITR award 0112628. We thank Dennis Brylow, Mayur Naik, James Rose, Vidyut Samanta, and Matthew Wallace for many helpful discussions.

A Proof of Theorem 2

Lemma 3. (Safe-Guarantee, Statements) *If* $\overline{\tau}, \widehat{imr} \ \vdash_K \ s \ : \ \widehat{imr}'$, *then* $safe(\overline{\tau}, \widehat{imr}, K)$.

Proof. By induction on the derivation of $\overline{\tau}, \widehat{imr} \ \vdash_K \ s \ : \ \widehat{imr}'$; we omit the details.

Lemma 4. (Safe-Guarantee, Handlers) *If* $\overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}'$, *then* $safe(\overline{\tau}, \widehat{imr}, K)$.

Proof. By induction on the derivation of $\overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}'$, using Lemma 3; we omit the details.

Lemma 5. (Safe-Guarantee, Main) *If* $\overline{\tau}, \widehat{imr} \ \vdash_K \ m$, *then* $safe(\overline{\tau}, \widehat{imr}, K)$.

Proof. By induction on the derivation of $\overline{\tau}, \widehat{imr} \ \vdash_K \ m$, using Lemma 3; we omit the details.

Lemma 6. (Safe-Weakening) *If* $K_1 \leq K_2$ *and* $safe(\overline{\tau}, \widehat{imr}, K_1)$, *then* $safe(\overline{\tau}, \widehat{imr}, K_2)$.

Proof. From $K_1 \leq K_2$ and

$$safe(\overline{\tau}, \widehat{imr}, K_1) = \left[\begin{array}{l} \forall i \in 1 \ldots n \\ \mathsf{if}\ enabled(\widehat{imr}, i) \\ \mathsf{then,\ whenever}\ \overline{\tau}(i) = \ldots \bigwedge (\widehat{imr} \xrightarrow{\delta} \widehat{imr}') \ \bigwedge \ldots, \\ \mathsf{we\ have}\ \widehat{imr}' \leq \widehat{imr} \ \wedge \ \delta + 1 \leq K_1 \end{array}\right]$$

we have

$$\left[\begin{array}{l} \forall i \in 1 \ldots n \\ \mathsf{if}\ enabled(\widehat{imr}, i) \\ \mathsf{then,\ whenever}\ \overline{\tau}(i) = \ldots \bigwedge (\widehat{imr} \xrightarrow{\delta} \widehat{imr}') \ \bigwedge \ldots, \\ \mathsf{we\ have}\ \widehat{imr}' \leq \widehat{imr} \ \wedge \ \delta + 1 \leq K_2 \end{array}\right]$$

that is, $safe(\overline{\tau}, \widehat{imr}, K_2)$.

Lemma 7. (K-Weakening, Statements) *If* $K_1 \leq K_2$ *and* $\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s \ : \ \widehat{imr}'$, *then* $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s \ : \ \widehat{imr}'$.

Proof. We proceed by induction on the derivation of $\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s \ : \ \widehat{imr}'$. There are six subcases depending on which one of Rules (19)–(24) was the last one used in the derivation of $\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s \ : \ \widehat{imr}'$.

- Rule (19). We have

$$\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ x = e \ : \ \widehat{imr} \quad \Big[safe(\overline{\tau}, \widehat{imr}, K_1)\Big].$$

  From $K_1 \leq K_2$, $safe(\overline{\tau}, \widehat{imr}, K_1)$, and Lemma 6, we have $safe(\overline{\tau}, \widehat{imr}, K_2)$. Hence, $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ x = e \ : \ \widehat{imr}$.
- Rule (20). The proof is similar to that for Rule (19).
- Rule (21). The proof is similar to that for Rule (19).
- Rule (22). We have

$$\frac{\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s_1 \ : \ \widehat{imr}' \quad \overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s_2 \ : \ \widehat{imr}'}{\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ \mathsf{if0} \ (x) \ s_1 \ \mathsf{else} \ s_2 \ : \ \widehat{imr}'} \quad \Big[safe(\overline{\tau}, \widehat{imr}, K_1)\Big].$$

  From the induction hypothesis, we have $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s_1 \ : \ \widehat{imr}'$ and $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s_2 \ : \ \widehat{imr}'$. From $K_1 \leq K_2$, $safe(\overline{\tau}, \widehat{imr}, K_1)$, and Lemma 6, we have $safe(\overline{\tau}, \widehat{imr}, K_2)$. Hence, $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ \mathsf{if0} \ (x) \ s_1 \ \mathsf{else} \ s_2 \ : \ \widehat{imr}'$.
- Rule (23). We have

$$\frac{\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s_1 \ : \ \widehat{imr}_1 \quad \overline{\tau}, \widehat{imr}_1 \ \vdash_{K_1} \ s_2 \ : \ \widehat{imr}_2}{\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s_1; s_2 \ : \ \widehat{imr}_2}.$$

  From the induction hypothesis, we have $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s_1 \ : \ \widehat{imr}_1$ and $\overline{\tau}, \widehat{imr}_1 \ \vdash_{K_2} \ s_2 \ : \ \widehat{imr}_2$. Hence, $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s_1; s_2 \ : \ \widehat{imr}_2$.
- Rule (24). The proof is similar to that for Rule (19).

Lemma 8. (K-Weakening, Handlers) *If* $K_1 \leq K_2$ *and* $\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ h \ : \ \widehat{imr}'$, *then* $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ h \ : \ \widehat{imr}'$.

Proof. We proceed by induction on the derivation of $\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ h \ : \ \widehat{imr}'$. There are two subcases depending on which one of Rules (17)–(18) was the last one used in the derivation of $\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ h \ : \ \widehat{imr}'$.

- Rule (17). We have

$$\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ \mathsf{iret} \ : \ \widehat{imr} \vee \mathsf{t}_0 \quad \Big[safe(\overline{\tau}, \widehat{imr}, K_1)\Big].$$

  From $K_1 \leq K_2$, $safe(\overline{\tau}, \widehat{imr}, K_1)$, and Lemma 6, we have $safe(\overline{\tau}, \widehat{imr}, K_2)$. Therefore, we have

$$\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ \mathsf{iret} \ : \ \widehat{imr} \vee \mathsf{t}_0.$$

– Rule (18). We have

$$\frac{\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s \ : \ \widehat{imr}' \qquad \overline{\tau}, \widehat{imr}' \ \vdash_{K_1} \ h \ : \ \widehat{imr}''}{\overline{\tau}, \widehat{imr} \ \vdash_{K_1} \ s; h \ : \ \widehat{imr}''}.$$

From Lemma 7, we have

$$\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s \ : \ \widehat{imr}'.$$

From the induction hypothesis, we have

$$\overline{\tau}, \widehat{imr}' \ \vdash_{K_2} \ h \ : \ \widehat{imr}''.$$

From $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s \ : \ \widehat{imr}'$ and $\overline{\tau}, \widehat{imr}' \ \vdash_{K_2} \ h \ : \ \widehat{imr}''$, we can use Rule (18) to derive $\overline{\tau}, \widehat{imr} \ \vdash_{K_2} \ s; h \ : \ \widehat{imr}''$.

We can now prove Theorem 2, which we restate here:

Suppose P is a consistent program state. If $\overline{\tau} \vdash_K P$, $K \geq 0$, and $P \rightarrow P'$, then $\overline{\tau} \vdash_{K'} P'$ and $K' \geq 0$, where $K' = K + |P.stk| - |P'.stk|$.

Proof. There are nine cases depending on which one of Rules (1)–(9) was used to derive $P \rightarrow P'$.

– Rule (1). We have $\langle \overline{\tau}, R, imr, \sigma, a \rangle \rightarrow \langle \overline{h}, R, imr \wedge \mathsf{t}_0^\bullet, a :: \sigma, \overline{h}(i) \rangle$ and $enabled(imr, i)$. Since P is consistent, there are two subcases.
Subcase 1: We have $P = \langle \overline{h}, R, imr, \mathsf{nil}, m \rangle$ and $P' = \langle \overline{h}, R, imr \wedge \mathsf{t}_0^\bullet, m :: \mathsf{nil}, \overline{h}(i) \rangle$. From $\overline{\tau} \vdash_K P$ and Rule (10), we have the derivation:

$$\frac{\overline{\tau} \ \vdash \ \overline{h} \ : \ \overline{\tau} \qquad imr \leq \widehat{imr} \qquad \overline{\tau}, \widehat{imr} \ \vdash_K \ m}{\overline{\tau} \ \vdash_K \ \langle \overline{h}, R, imr, \mathsf{nil}, m \rangle}.$$

From $\overline{\tau}, \widehat{imr} \ \vdash_K \ m$, and Lemma 5, we have that:

$$safe(\overline{\tau}, \widehat{imr}, K) = \begin{bmatrix} \forall i \in 1 \ldots n \\ \mathsf{if} \ enabled(\widehat{imr}, i) \\ \mathsf{then, whenever} \ \overline{\tau}(i) = \ldots \bigwedge (\widehat{imr} \xrightarrow{\delta} \widehat{imr}') \ \bigwedge \ldots, \\ \mathsf{we \ have} \ \widehat{imr}' \leq \widehat{imr} \ \wedge \ \delta + 1 \leq K \end{bmatrix}$$

is true. From $safe(\overline{\tau}, \widehat{imr}, K)$ and $enabled(\widehat{imr}, i)$, it follows that:

$$\overline{\tau}(i) = \ldots \bigwedge (\widehat{imr} \xrightarrow{\delta} \widehat{imr}') \bigwedge \ldots \qquad \widehat{imr}' \leq \widehat{imr} \qquad \delta + 1 \leq K.$$

From $\overline{\tau} \ \vdash \ \overline{h} \ : \ \overline{\tau}$ and Rule (14), we have $\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_\delta \ h_i \ : \ \widehat{imr}'$. From $\delta \leq K - 1$, $\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_\delta \ h_i \ : \ \widehat{imr}'$, and Lemma 8, we have

$$\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_{K-1} \ h_i \ : \ \widehat{imr}'.$$

From $\overline{\tau}, \widehat{imr} \ \vdash_K \ m$ and Rule (12), we have $\overline{\tau}, \widehat{imr} \ \vdash_{K-1} \ m :: \mathsf{nil}$. From $\overline{\tau} \ \vdash \ \overline{h} \ : \ \overline{\tau}$, $imr \wedge \mathsf{t}_0^\bullet \leq \widehat{imr} \wedge \mathsf{t}_0^\bullet$, $\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_{K-1} \ h_i \ : \ \widehat{imr}'$, $\widehat{imr}' \leq \widehat{imr}$, $\overline{\tau}, \widehat{imr} \ \vdash_{K-1} \ m :: \mathsf{nil}$, and $K' = K + |P.stk| - |P'.stk| = K - 1 \geq \delta \geq 0$, we can use Rule (11) to derive $\overline{\tau} \ \vdash_{K'} \ P'$.
Subcase 2: We have $P = \langle \overline{h}, R, imr, \sigma, h \rangle$, $P' = \langle \overline{h}, R, imr \wedge \mathsf{t}_0^\bullet, h :: \sigma, \overline{h}(i) \rangle$. From $\overline{\tau} \vdash_K P$ and Rule (11), we have the derivation:

$$\frac{\overline{\tau} \ \vdash \ \overline{h} \ : \ \overline{\tau} \quad imr \leq \widehat{imr} \qquad \overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}_r \quad \widehat{imr}_r \leq \widehat{imr}_b \quad \overline{\tau}, \widehat{imr}_b \ \vdash_K \ \sigma}{\overline{\tau} \ \vdash_K \ \langle \overline{h}, R, imr, \sigma, h \rangle}.$$

From $\overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}_r$, and Lemma 4, we have that

$$safe(\overline{\tau}, \widehat{imr}, K) = \begin{bmatrix} \forall i \in 1 \ldots n \\ \mathsf{if} \ enabled(\widehat{imr}, i) \\ \mathsf{then, whenever} \ \overline{\tau}(i) = \ldots \bigwedge \ (\widehat{imr} \xrightarrow{\delta} \widehat{imr}') \ \bigwedge \ldots, \\ \mathsf{we \ have} \ \widehat{imr}' \leq \widehat{imr} \ \wedge \ \delta + 1 \leq K \end{bmatrix}$$

is true. From $safe(\overline{\tau}, \widehat{imr}, K)$ and $enabled(\widehat{imr}, i)$, it follows that

$$\overline{\tau}(i) = \ldots \bigwedge (\widehat{imr} \xrightarrow{\delta} \widehat{imr}') \bigwedge \ldots \qquad \widehat{imr}' \leq \widehat{imr} \qquad \delta + 1 \leq K.$$

From $\overline{\tau} \ \vdash \ \overline{h} \ : \ \overline{\tau}$ and Rule (14), we have $\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_\delta \ h_i \ : \ \widehat{imr}'$. From $\delta \leq K - 1$, $\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_\delta \ h_i \ : \ \widehat{imr}'$, and Lemma 8, we have

$$\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_{K-1} \ h_i \ : \ \widehat{imr}'.$$

From $\overline{\tau}, \widehat{imr} \ \vdash_K \ h \ : \ \widehat{imr}_r$, $\widehat{imr}_r \leq \widehat{imr}_b$, $\overline{\tau}, \widehat{imr}_b \ \vdash_K \ \sigma$, and Rule (13), we have

$$\overline{\tau}, \widehat{imr} \ \vdash_{K-1} \ h :: \sigma.$$

From $\overline{\tau} \ \vdash \ \overline{h} \ : \ \overline{\tau}$, $imr \wedge \mathsf{t}_0^\bullet \leq \widehat{imr} \wedge \mathsf{t}_0^\bullet$, $\overline{\tau}, \widehat{imr} \wedge \mathsf{t}_0^\bullet \ \vdash_{K-1} \ h_i \ : \ \widehat{imr}'$, $\widehat{imr}' \leq \widehat{imr}$, $\overline{\tau}, \widehat{imr} \ \vdash_{K-1} \ h :: \sigma$, and $K' = K + |P.stk| - |P'.stk| = K - 1 \geq \delta \geq 0$, we can use Rule (11) to derive $\overline{\tau} \ \vdash_{K'} \ P'$.

– Rule (2). We have $\langle \overline{h}, R, imr, \sigma, \mathsf{iret} \rangle \rightarrow \langle \overline{h}, R, imr \vee \mathsf{t}_0, \sigma', a \rangle$, and $\sigma = a :: \sigma'$. Since P is consistent, there are two subcases.
Subcase 1: We have $P = \langle \overline{h}, R, imr, m :: \mathsf{nil}, \mathsf{iret} \rangle$ and
$P' = \langle \overline{h}, R, imr \vee \mathsf{t}_0, \mathsf{nil}, m \rangle$. From $\overline{\tau} \vdash_K P$, Rule (11), and Rule (12), we have the derivation:

$$\frac{\overline{\tau} \ \vdash \ \overline{h} \ : \ \overline{\tau} \quad imr \leq \widehat{imr} \qquad \overline{\tau}, \widehat{imr} \ \vdash_K \ \mathsf{iret} \ : \ \widehat{imr} \vee \mathsf{t}_0 \quad \widehat{imr} \vee \mathsf{t}_0 \leq \widehat{imr}_b \quad \dfrac{\overline{\tau}, \widehat{imr}_b \ \vdash_{K+1} \ m}{\overline{\tau}, \widehat{imr}_b \ \vdash_K \ m :: \mathsf{nil}}}{\overline{\tau} \ \vdash_K \ \langle \overline{h}, R, imr, m :: \mathsf{nil}, \mathsf{iret} \rangle}.$$

From $\overline{\tau} \vdash \overline{h} : \overline{\tau}$, $imr \vee \mathsf{t}_0 \leq \widehat{imr} \vee \mathsf{t}_0 \leq \widehat{imr}_b$, $\overline{\tau}, \widehat{imr}_b \vdash_{K+1} m$, and $K' = K + |P.stk| - |P'.stk| = K+1$, we can use Rule (10) to derive $\overline{\tau} \vdash_{K'} P'$.
Subcase 2: We have $P = \langle \overline{h}, R, imr, h^k :: \sigma', \mathsf{iret} \rangle$ and
$P' = \langle \overline{h}, R, imr \vee \mathsf{t}_0, \sigma', h^k \rangle$. From $\overline{\tau} \vdash_K P$, Rule (11), and Rule (13), we have the derivation:

$$\frac{\overline{\tau} \vdash \overline{h} : \overline{\tau} \quad imr \leq \widehat{imr} \qquad \overline{\tau}, \widehat{imr} \vdash_K \mathsf{iret} : \widehat{imr} \vee \mathsf{t}_0 \quad \widehat{imr} \vee \mathsf{t}_0 \leq \widehat{imr}_b \quad \overline{\tau}, \widehat{imr}_b \vdash_K h^k :: \sigma'}{\overline{\tau} \vdash_K \langle \overline{h}, R, imr, h^k :: \sigma', \mathsf{iret} \rangle}.$$

where $\overline{\tau}, \widehat{imr}_b \vdash_K h^k :: \sigma'$ is derived as follows:

$$\frac{\overline{\tau}, \widehat{imr}_b \vdash_{K+1} h^k : \widehat{imr}_r^k \quad \widehat{imr}_r^k \leq \widehat{imr}_b^k \quad \overline{\tau}, \widehat{imr}_b^k \vdash_{K+1} \sigma}{\overline{\tau}, \widehat{imr}_b \vdash_K h^k :: \sigma}.$$

From $\overline{\tau} \vdash \overline{h} : \overline{\tau}$, $imr \vee \mathsf{t}_0 \leq \widehat{imr} \vee \mathsf{t}_0 \leq \widehat{imr}_b$, $\overline{\tau}, \widehat{imr}_b \vdash_{K+1} h^k : \widehat{imr}_r^k$, $\widehat{imr}_r^k \leq \widehat{imr}_b^k$, $\overline{\tau}, \widehat{imr}_b^k \vdash_{K+1} \sigma$, and $K' = K + |P.stk| - |P'.stk| = K + 1$ we can use Rule (11) to derive $\overline{\tau} \vdash_{K'} P'$.

– Rule (3). We have $\langle \overline{h}, R, imr, \mathsf{nil}, \mathsf{loop}\ s \rangle \rightarrow \langle \overline{h}, R, imr, \mathsf{nil}, s; \mathsf{loop}\ s \rangle$. From $\overline{\tau} \vdash_K P$, Rule (10), and Rule (15), we have the derivation:

$$\frac{\overline{\tau} \vdash \overline{h} : \overline{\tau} \qquad imr \leq \widehat{imr} \qquad \dfrac{\overline{\tau}, \widehat{imr} \vdash_K s : \widehat{imr}}{\overline{\tau}, \widehat{imr} \vdash_K \mathsf{loop}\ s}}{\overline{\tau} \vdash_K \langle \overline{h}, R, imr, \mathsf{nil}, \mathsf{loop}\ s \rangle}.$$

From $\overline{\tau}, \widehat{imr} \vdash_K s : \widehat{imr}$, $\overline{\tau}, \widehat{imr} \vdash_K \mathsf{loop}\ s$, and Rule (16) we have $\overline{\tau}, \widehat{imr} \vdash_K s; \mathsf{loop}\ s$. From $\overline{\tau} \vdash \overline{h} : \overline{\tau}$, $imr \leq \widehat{imr}$, $\overline{\tau}, \widehat{imr} \vdash_K s; \mathsf{loop}\ s$, and $K' = K + |P.stk| - |P'.stk| = K$, we can use Rule (10) to derive $\overline{\tau} \vdash_{K'} P'$.

– Rule (4). We have $\langle \overline{h}, R, imr, \sigma, x = e; a \rangle \rightarrow \langle \overline{h}, R\{x \mapsto eval_R(e)\}, imr, \sigma, a \rangle$. Since P is consistent, there are two subcases.
Subcase 1: $P = \langle \overline{h}, R, imr, \mathsf{nil}, x = e; m \rangle$ and
$P' = \langle \overline{h}, R\{x \mapsto eval_R(e)\}, imr, \mathsf{nil}, m \rangle$.
From $\overline{\tau} \vdash_K P$, Rule (10), and Rule (16), we have the derivation:

$$\frac{\overline{\tau} \vdash \overline{h} : \overline{\tau} \quad imr \leq \widehat{imr} \qquad \dfrac{\overline{\tau}, \widehat{imr} \vdash_K x = e : \widehat{imr} \quad \overline{\tau}, \widehat{imr} \vdash_K m}{\overline{\tau}, \widehat{imr} \vdash_K x = e; m}}{\overline{\tau} \vdash_K \langle \overline{h}, R, imr, \mathsf{nil}, x = e; m \rangle}.$$

From $\overline{\tau} \vdash \overline{h} : \overline{\tau}$, $imr \leq \widehat{imr}$, $\overline{\tau}, \widehat{imr} \vdash_K m$, and $K' = K + |P.stk| - |P'.stk| = K$, we can use Rule (10) to derive $\overline{\tau} \vdash_{K'} P'$.
Subcase 2: $P = \langle \overline{h}, R, imr, \sigma, x = e; h \rangle$ and
$P' = \langle \overline{h}, R\{x \mapsto eval_R(e)\}, imr, \sigma, h \rangle$. From $\overline{\tau} \vdash_K P$, Rule (11), and

Rule (18), we have the derivation:

$$\frac{\overline{\tau} \vdash \overline{h} : \overline{\tau} \quad imr \leq \widehat{imr} \qquad \dfrac{\overline{\tau}, \widehat{imr} \vdash_K x = e : \widehat{imr} \quad \overline{\tau}, \widehat{imr} \vdash_K h : \widehat{imr}_r}{\overline{\tau}, \widehat{imr} \vdash_K x = e; h : \widehat{imr}_r} \quad \widehat{imr}_r \leq \widehat{imr}_b \quad \overline{\tau}, \widehat{imr}_b \vdash_K \sigma}{\overline{\tau} \vdash_K \langle \overline{h}, R, imr, \sigma, x = e; h \rangle}.$$

From $\overline{\tau} \vdash \overline{h} : \overline{\tau}$, $imr \leq \widehat{imr}$, $\overline{\tau}, \widehat{imr} \vdash_K h : \widehat{imr}_r$, $\widehat{imr}_r \leq \widehat{imr}_b$, $\overline{\tau}, \widehat{imr}_b \vdash_K \sigma$, and $K' = K + |P.stk| - |P'.stk| = K$, we can use Rule (11) to derive $\overline{\tau} \vdash_{K'} P'$.

– Rules (5)–(9). The proofs are similar to that for Rule (4); we omit the details.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1981.
3. Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proceedings of ICSE'01, 23rd International Conference on Software Engineering*, pages 47–56, Toronto, May 2001.
4. Luca Cardelli and Andrew D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures*, pages 140–155. Springer-Verlag (*LNCS* 1378), 1998.
5. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda-calculus semantics. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 535–560. Academic Press, 1980.
6. J. Roger Hindley. Types with intersection: An introduction. *Formal Aspects of Computing*, 4:470–486, 1991.
7. R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag (*LNCS* 92), 1980.
8. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Compuation*, 100(1):1–77, September 1992.
9. Mayur Naik and Jens Palsberg. Compiling with code-size constraints. In *LCTES'02, Languages, Compilers, and Tools for Embedded Systems joint with SCOPES'02, Software and Compilers for Embedded Systems*, June 2002.
10. Jens Palsberg. Type-based analysis and applications. In *Proceedings of PASTE'01, ACM Workshop on Program Analysis for Software Tools*, pages 20–27, June 2001.
11. Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001.
12. Jens Palsberg and Matthew Wallace. Reverse engineering of real-time assembly code. Manuscript, 2002.
13. Andreas Podelski. Model checking as constraint solving. In *Proceedings of SAS'00, International Static Analysis Symposium*, pages 22–37. Springer-Verlag (*LNCS* 1824), 2000.
14. Wayne Wolf. *Computers as Components, Principles of Embedded Computing System Design*. Morgan Kaufman Publishers, 2000.