C_0, an Imperative Programming Language for Novice Computer Scientists

Rob Arnold

CMU-CS-10-145

December 2010

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, Chair
Mark Stehlik

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This work was partially supported by the Computational Thinking Center at Carnegie Mellon University, sponsored by Microsoft Research.

Keywords: c, programming language, education

Abstract

The primary goal of a computer science course on data structures and algorithms is to educate students on the fundamental abstract concepts and expose them to concrete examples using programming assignments. For these assignments, the programming language should not dominate the assignment but instead complement it so that students spend most of their time engaged in learning the course material. Choosing an appropriate programming language requires consideration not just of the course material and the programs students will write but also their prior programming experience and the position of the course in the course sequence for their degree program.

We have developed a new programming language, C_0, designed for Carnegie Mellon's new Principles of Imperative Programming course. C_0 is almost a subset of C, eschewing complex semantics and undefined or unspecified behaviors in favor of simple semantics and formally specified behavior. It provides features such as garbage collection and array bounds checks to aid students in developing and reasoning about their programs. In addition to the compiler and runtime, we have also developed a small set of libraries for students to use in their assignments.

Acknowledgments

There are many people who supported me in my work.

First, I would like to thank my advisor Frank Pfenning for his guidance, feedback, and funding this past year. I would also like to thank Mark Stehlik for serving on my thesis committee.

Rob Simmons, William Lovas, and Roger Wolff were willing at all times to discuss all ideas and implementation ideas. Michael Sullivan, Max Buevich, and Dan Schafer also gave me good feedback during the development of C_0. Their feedback was invaluable in helping me in this endeavor.

I want to extend a special thanks to Deborah Cavlovich for helping me sort out all the administrative details and issues.

Finally I want to thank my parents, sister and grandmother for their support of my education over the years.

Contents

Chapter 1

Introduction

Designing a new language is not something one does lightly. There are hundreds of existing languages to choose from with many different properties so it seems reasonable to assume that there must exist some language suitable for teaching introductory data structures. Many of these langauges were not designed with pedagogical purposes in mind. They may have historical or designer quirks and features that no longer (or never did) make sense. They are not usually designed with novice programmers in mind but to accomplish some particular task or set of tasks. In the case of Carnegie Mellon's introductory data structures course, a suitable language would need to be easily accessible by programmers with minimal experience and it must be able to adequately express the data structures used in the course.

1.1 Motivation

Carnegie Mellon is revising its early undergraduate curriculum. In particular, they are bringing Jeanette Wing's [10] idea of computational thinking to both majors and nonmajors, increase focus on software reliability and integrate parallelism into the core curriculum. These changes involve restructuring the course sequence and content.

The first course, 15-110, in the sequence is intended for nonmajors and majors with no programming experience. It is intended to be an introductory computer science course containing the basic principles including an emphasis on computational thinking. The next course, 15-122, focuses on writing imperative programs with fundamental data structures and algorithms and reasoning about them. Students also take a functional programming course, 15-150, which focuses on working with immutable data and the concepts of modularity and abstraction. Following 15-122 and 15-150, students take 15-210 where they continue to learn more advanced data structures as in 15-122 though with an additional emphasis on parallel programming. 15-213 remains an introductory systems course and 15-214 focuses on designing large-scale systems and includes content on object-oriented programming and distributed concurrency.

For this thesis, we focus on 15-122 and the choice of programming language for its assignments.

1.2 Language Requirements

To formulate the requirements, we must first examine the types of programs that students will be assigned to write for the course. The assignments are intended to assess and educate students according to the learning goals of the course.

The majority of students taking 15-122 will have scored well on the AP test which currently uses Java. Those that have not will have taken a semester-long basic programming course taught using Python as the primary language for assignments. Though students should be able to write basic programs upon entry to this course, we do not assume that they will have had much experience in debugging or using the intermediate or advanced features of either language. This course is intended to instruct students in the basic data structures and algorithms used in computer science. This includes:

- Stacks and Queues
- Binary Search
- Sorting
- Priority queues
- Graph traversal / shortest path
- Balanced binary trees
- Game tree search
- String searching

By the end of the course, students should be able to:

- For the aforementioned data structures, explain the basic performance characteristics for various operations and write imperative code that implements these data structures.
- Using their knowledge of the basic performance characteristics, choose appropriate data structures to solve common computer science problems.
- Be able to reason about their programs using invariants, assertions, preconditions and post-conditions.

Based on the learning goals of the course and its place in the course sequence, I formulated the following design requirements for the language and compiler/interpreter:

Simple as possible The languages used in industry are aimed at professionals and problems in software engineering, not necessarily computer science. Many of these languages impose design requirements which, though often valuable for software engineering purposes, are unnecessary cognitive overhead for students to learn and comprehend. Explaining these features takes time away from teaching the intended topics of the course.

Friendly error messages Compilers are also not known for giving helpful or useful error messages to novices. Most SML and C++ compilers are notorious in the academic and industrial worlds for giving cryptic error messages that only begin to make sense once a deep understanding of the language is obtained.

Similar enough to languages used in industry Students often want to search for internships in the summer following their first year as an undergraduate. It does not help them in their

search if they have learned an obscure or irrelevant programming language in this course. Thus our choice of language must be or be similar enough to an existing language that is used in industry so that students can legitimately claim, based on their coursework, that they have a marketable skill.

1.2.1 Simplicity

"Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher" - Antoine de Saint Exupéry

Some languages have low cognitive overhead because they can be taught in layers, starting with simple concepts and introducing new features that build upon these basics. Algebra is often taught this way to high school students by starting with the concepts of variables and the basic axioms and gradually moving onto harder problems. Carnegie Mellon's sophomore-level Principles of Programming course teaches Standard ML this way: the first assignment has students writing simple values and expressions in the language and then moves on to functions and more complicated data structures like lists and trees[1]. Though these sorts of languages are often amenable to including an interpreter, it is by no means a requirement.

Though languages like Java are quite popularly used in introductory data structures courses, that does not mean they are a good choice. Programming even the tiniest bit in Java requires using classes, objects. For basic IO, packages are needed. Take the example of writing a linked list type and calculating the length of a one element instance. The Java[7] version (1.2.1) requires 3 classes and the explanation of member variables and methods. Compare this to the conceptual size of comparable code in C[11] (1.2.1) and Standard ML[13] (1.2.1) which just require understanding functions (and in ML's case, a recursively defined one - more on this later). Notice in particular how the Java definition requires a separate Node and List class because methods such as `length()` cannot be invoked on null objects[2].

Looking at the course content, all that is needed to express the mathematical ideas is relatively simple data structures and functions to modify and query them. Furthermore, object-oriented programming is covered in a subsequent couse. Thus our desired language need not support objects and should not require them.

Functional programming languages like SML, Haskell and F# are generally still problematic for students new to programming. The majority (if not all) of students taking 15-122 will have been taught a language like Python or Java and likely have no experience working in a functional programming language. Often the basic programming tasks in these functional languages require defining simple functions recursively as in 1.2.1 or using functions as first class values which are difficult concepts for novices to understand. Though understanding these concepts is important for their overall education, they are not key towards understanding the material in the course. The functional programming course, 15-150, will cover this material and functional programming is also heavily used in the subsequent parallel data structures and algorithms course, 15-210.

It is also often the case that many algorithms are given in terms of an imperative pseudo-code

[1] Also see `tryhaskell.org` for a compressed approach with Haskell

[2] In C++ you can get away with having a single class in most implementations since non-virtual methods don't require a non-null **`this`**

Simple linked list written in Java

```
class Node { int data; Node next; }

class List {
  Node head;

  int length() {
    int len = 0;
    Node p = head;
    while(p != null) {
      len++;
      p = p.next;
    }
    return len;
  }
}
```

Simple linked list written in C

```
#include <stddef.h>

struct list {
  int data;
  struct list *next;
}

int length(struct list *p) {
  int l = 0;
  while (p) {
    l++;
    p = p->next;
  }
  return l;
}
```

Simple linked list written in SML

```
datatype 'a list =
    Nil
  | Data of 'a * 'a list

fun length Nil = 0
  | length (Data (hd,tl)) = 1 + length tl
```

with mutable variables and loops. These features are desirable to have in our language as well to minimize the student's burden of transferring their knowledge into code.

1.2.2 Friendly Error Messages

Error messages are a frequent source of frustration for introductory computer science students. Even professionals find them unhelpful or even misleading. Though some work has been done recently in production compilers [3], many compilers produce technically worded errors which are sometimes not even local to the actual problem! Thus the compiler for 15-122 should produce friendly error messages whenever possible. They should point to the precise source of the problem and offer suggestions to fixes. Note that we don't want to automatically fix these errors - the compiler may not always be able to choose the right fix to apply and fixing problems in the source is an important skill for students to have. For some languages, this can be rather difficult due to extensive elaboration or other transformations of the source prior to typechecking.

1.2.3 Similarity to a Real Industrial Language

Carnegie Mellon undergraduates often take summer internships and end up taking a fulltime job in the software industry upon graduation. While one could argue that the purpose of a university education is not to be job training, it is important that motivated freshmen should be able to obtain an internship after their first year. It helps tremendously to be able to claim experience with a well-known language used in industry. Though ideally the language used for the assignments would be an industrial language, for previously mentioned reasons most, if not all, of the popular languages in industry are so far unsuitable. Going down the path of picking an unpopular language requires an eventual transition to a real industrial language. The more similarities between our language and some existing one allows for a easier transition for students. The difficulty lies in balancing this requirement with the aforementioned ones, in particular simplicity.

Carnegie Mellon's systems courses heavily use C for their assignments. Given that 15-122 is a prerequisite for these courses and C's heavy use in industry, it is the natural choice. Using a C-like language early in the curriculum may have the added benefit of increasing students' proficiency with the language due to the extra time they spend using the language.

This thesis is organized as follows: Chapter 2 presents the language with concrete syntax and a prose description of the semantics. Chapter 3 introduces the abstract syntax and presents the progress and preservation properties. Chapter 4 covers the implementation details of our reference compiler and runtime. Chapter 5 introduces the standard libraries used by assignments in the course. Chapter 6 covers the differences between C_0 and C and presents ideas for future work.

[3] http://clang.llvm.org/diagnostics.html

Chapter 2

Language Description

C_0 is a statically typed imperative programming language. It strongly resembles C in syntax and semantics. In designing it, we tried to stay close to C's syntax and semantics in the hopes that students could simply include a header file in each C_0 program and it would be accepted by a C compiler. Unfortunately, some of C_0's semantics differ from those of C and in these cases, using C's syntax would be a confusing choice for students when it came time for them to learn C. Thus we changed the syntax for these constructs to the syntax of modern languages with similar semantics. This made it impossible for there to be a header that will allow C_0 programs to be compiled as C. However, the transformation remains a simple operation with a trivial `sed` script or doing it by hand for small programs.

2.1 Basic data types and expressions

Expressions are evaluated in left-to-right order (that is, a post-order traversal of the abstract syntax tree). Expressions come in many forms. The common ones are:

expression:
- *constant*
- *variable-name*
- *expression binop expression*
- *monop expression*

Expressions in C_0 are not pure; they may have side effects such as modifying memory or raising exceptions.

2.1.1 Booleans

C_0 has a few basic builtin types. The simplest of these is the boolean type. Similar to C++, Java and C#, we write the type as **`bool`** and its two values are **`true`** and **`false`**.

bool-constant:

```
      true
      false
```
constant:
      bool-constant
ty:
```
      bool
```

```
bool b = true || false && !(x == y);
```

The usual set of basic boolean operations are provided for logical and (`&&`), logical or (`||`), logical not (`!`), and equality/inequality (`==`,`!=`). As in many languages, these operators can "short circuit" their evaluation when the value from evaluating the left hand side determines the expression's value.

There is also a conditional expression as in C and Java: `e ? a : b` where `e` is an expression that yields a boolean. If it evaluates to **`true`** then `a` is evaluated to be the result of the ternary expression. Otherwise `b` is evaluted to be the result of the ternary expression.

2.1.2 Integers

C_0 has a single numerical type **`int`** which has a range of $[-2^{31}, 2^{31})$. It is represented as a signed 32 bit two's complement integer. We felt that it would be more helpful to specify the exact size, numerical range and representation of the integer type. In addition, having a single numerical type greatly simplifies the language and cognitive requirements for students. It is an unfortunate limitation that the numerical type is integral as there are a number of interesting data structures and projects that would be much easier to represent as some non-integral type like IEEE 754 floating point or .NET's decimal type. The problem with these data types however is that they are only approximations of real numbers so students will need to be made aware of their underlying implementation and properties. These properties are non-trivial and the cognitive overhead required for students to understand them is not worth the time and effort when there are already many interesting data structures within the scope of the course that work well with integers. Furthermore, these numerical types and their representations are covered in the subsequent introductory systems course, 15-213.

Integer literals are written as a sequence of digits, either hexadecimal digits with an `0x` prefix as in C or decimal digits with an optional `-` prefix to indicate negative numbers. Basic arithmetic operations include modular addition, subtraction, negation, multiplication, division and remainder operations (`+ - * / %`). For the division and remainder operations, if the divisor is 0 or additionally for negation if the result does not fall into the range of the **`int`** type, an exception is triggered which results in the termination of the program. The division and remainder operations round towards 0.

ty:
```
      int
```
constant:

integer-constant

```
int x = 0x123 + -456 * 20;
```

Basic bitwise operations are also supported: bitwise and (`&`), bitwise or (`|`), bitwise xor (`^`), bitwise inversion (`~`). Arithmetic shift operations are also supported with the `<<` and `>>` binary operators. Only the least significant 5 bits of the shift's two's complement binary representation are used to determine the number of bits that are shifted. These 5 bits are treated as an unsigned integer. For example, the expression `~19 << 2` evaluates to `-80`.

2.1.3 Strings and Characters

C_0 provides a basic opaque type **`char`** for representing characters. The exact representation is not defined however they may be specified by a printable ASCII character enclosed in single quotes such as `'a'` and `'9'`. To accommodate some useful but not printable ASCII characters, the following escape sequences are also permitted: `\t` (tab), `\r` (return), `\b` (backspace), `\n` (newline), `\'` (single quote), `\"` (double quote) and `\0` (null).

C_0 also defines an opaque immutable string type **`string`** which contains a sequence of characters. There is no direct conversion between strings and characters however several library functions such as `string_charat` and `string_to_chararray` provide access to the characters. String literals are written as a sequence of ASCII characters or permitted **`char`** escape sequences (except `\0`) enclosed in double quotes. Strings may be compared using the library functions `string_equal` and `string_compare`.

```
string s = "Hello world";
```

ty:
    `char`
    `string`
constant:
    character-constant
    string-constant

2.1.4 Comparisons

Owing to its C heritage, C_0 provides equality and inequality operators as `==` and `!=` respectively for expressions of the same type. These operators are defined for the **`bool`**, **`int`** and **`char`** types as well as pointer types. Pointer types are checked only for reference equality. Comparisons on integers are also provided with the standard and self-explanatory concrete syntax from C (`< <= > >=`).

2.2 Variables & Statements

For now we will consider only three kinds of statements: blocks, variable declarations and assignments. As in C, blocks are defined as a possibly empty sequence of statements enclosed in curly braces (`{ }`). They are used for grouping related statements and determining variable scope. A block statement is executed by executing each of its statements in sequence.

compound-statement:
{ *statement** }

Variables are automatically managed memory locations that hold values. Variables are identified using a sequence of alphanumeric characters (though the identifier must start with a letter). Variable declarations such as `int x;` introduce a new variable that can be used by expressions and assignments. As in C99, a variable's scope is defined as the statement following the variable declaration to the end of the innermost block the declaration occured in.

declaration:
ty variable-name

Variable declaration restrictions:

1. A variable may only be referenced by expressions and statements inside its scope.
2. A variable may not be used in an expression if it cannot be statically verified that it has been assigned prior to use. That is, an assignment statement must dominate all uses in the control flow graph of the block containing the variable's scope. Loops are assumed to potentially never run for the purpose of this analysis.
3. Two variables of the same name may not have overlapping scopes.
4. The variable's type must be **`bool`**, **`int`**, a pointer type or an array type.

Ex:

```
int x;
{
  bool y;
  /* Error: x has not been assigned */
  y = x > 0;
  /* Error: y has not been assigned */
  y = !y;
}
/* Error: y is not in scope */
x = y > 0 ? 3 : 4;
int y;
y = x+2;
```

Assignment statements are used to update values stored in variables. The variable on the left hand side must be in scope.

simple-statement:
    lhs assign-op exp
atomic-expression:
    variable-name
lhs:
    atomic-expression

For convenience, there is also hybrid class of assignment statements that combine a binary operator with assignment for the common case where the left hand side of a binary operation is the left hand side of the assignment as well. For the assignment to variables, `x op=e` is simple syntactic sugar for `x = x op e`. Note that `*p op= e` cannot simply desugar to `*p = *p op e` when the evaluation of `p` may have side effects. For this reason, the left hand side is evaluated before the right hand side and only once.

There is a hybrid statement that is formed by combining a variable declaration with an assignment:

declaration:
    ty variable-name = *expression*

This is merely syntactic sugar: `T x = e;` $\equiv$ `T x; x = e;`

2.3 Functions

Functions are the primary mechanism for abstracting computations and reusing code. A function declares parameters which behave much like variable declarations at the top of the function's body except that they are guaranteed to be assigned some value upon entry to the body of the function. A function's body is executed when the function is applied to some arguments. The arguments are evaluated to values in the conventional left-to-right ordering. These values are then bound to the function's parameters in the same order and the body of the function is then executed in a new scope where only those parameters are bound. This means that functions cannot refer to variables declared in other functions. Thus each function can be typechecked independently from the definitions (but not declarations) of other functions.

compound-statement:
    { *statement** }
function-decl:
    ty function-name ($\mathit{function\text{-}params}_{opt}$)
function-def:
    function-decl compound-statement

Function application may occur as a top-level statement (that is, for its side-effects only) or as part of an expression, including arguments to other function applications. As an expression, function application must evaluate to some value. To indicate this value, a function uses the **`return`** statement. The **`return`** statement evaluates an expression to a value and then aborts execution of the callee's body, yielding the value as the result of the function application in the caller. To indicate what type of value is returned, functions declare a return type. All functions must return a value and must only return values of the declared return type. As with checking for assignment before use for variables, a conservative analysis is used so loops are not considered to necessarily be run, even in the trivial cases.

call-expression:
    lhs (*argument-list*$_{opt}$)
simple-statement:
    call-expression$_{opt}$

For example, suppose we have:

```
int triarea(int b, int h) {
  return b*h/2;
}
```

which is a function named `triarea` that returns a value of type **`int`** and has two parameters of type **`int`**, `b` and `h`. `triarea(3,4)` is an application of `triarea` to the expressions `3` and `4`. To evaluate this application, the arguments are evaluated into values which they already are, then they are bound in a new scope to the variables `b` and `h`. The return statement then evaluates `b*h/2` to a value (the integer represented by `6`) which then becomes the result of the original application.

Since the parameters are bound to values, a callee cannot affect the variable bindings in its caller's scope[1].

Functions may be named with any identifier using the same rules as for variables. Function and variable names may not conflict. Like C, a function needs to be declared before it can be used in another function.

```
int foo(int x) {
  x += 2;
  return x;
}

int bar(int y) {
  int x = y*2;
  int z = foo(x);
  /* x is still y*2 no matter what foo does. */
  return z+x;
}
```

[1]It can effect a change through the heap with more advanced types which refer to the heap such as pointers and arrays. The heap is discussed in section 2.5

2.4 Conditionals

C_0 offers a subset of the control flow statements in C that are found in other popular languages. The most basic one is **`if`** statement. The **`if`** statement allows execution the execution of a statement conditional upon some provided condition. It has two forms:

selection-statement:
`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*

Each time the **`if`** statement is executed, the condition is evaluated and the corresponding inner statement (if there is one) is executed. To repeatedly execute a statement until a condition no longer holds, the **`while`** statement can be used. It behaves just like the first form of an **`if`** statement that is re-executed until the condition is false.

iteration-statement:
`while (` *expression* `)` *statement*

```
x = 1;
while (x < y)
  x *= 2;
// At this point, it must be the case that x holds the smallest
// power of two greater than y
```

There are two additional statement types that are available for use within a loop: **`break`** and **`continue`**. Whenever a **`continue`** statement is executed, execution jumps to the top of the current innermost loop as if the body were done executing. Whenever a **`break`** is executed within a loop, the current innermost loop is ”broken” and execution resumes at whatever statement or condition follows that loop, as if the condition were **`false`**.

jump-statement:
`break ;`
`continue ;`

There is an additional form of loop, the **`for`** loop. It behaves much like C’s: there is an initial statement that is executed before the loop, a condition that is checked before every iteration, much like the while loop, and a post-loop-body statement which is executed after the body of the loop.

initializer:
declaration
simple-statement
iteration-statement:

`for` (*initializer* `;` $expression_{opt}$ `;` *simple-statement*) *statement*

Either of the two statements may be omitted and if the condition is omitted, it is assumed to be **`true`**. The initial statement may be a declaration but the post statement may not. To give the same behavior as C99/C++, the for loop is wrapped in an implicit block so the scope of a declaration in the initial statement is limited to the condition, body, and post statement.

As with **`while`** statements, the body is repeatedly executed while the condition is true. After the execution of the body, the post statement is run. The **`continue`** statement behaves slightly differently in a **`for`** loop; it jumps to the end of the body so that the post-loop-body statement is executed. The **`break`** statement behaviors exactly the same as in the **`while`** loop - it does not execute the post statement.

2.5 The Heap & Pointers

The heap is a separate persistent memory store, accessible from any function. Access to values in the heap is done only through references. References to values in the heap (addresses) are stored as pointer values. These pointer values may be copied around and passed to functions just as any other value is. They have a type dependent upon the type value they reference. This type is written as `T*` where `T` is the type of the value pointed to.

As with all local variables, pointers must be initialized before use. A pointer may be initialized to either a new value on the heap, the address of an existing value in the heap, or a special value called `NULL`. Attempts to reference the heap using `NULL` will abort execution. It exists solely to indicate an invalid reference.

New values on the heap are created with the `alloc` keyword like so:

expression:
`alloc` (*ty*)

Because it is difficult to determine if a value in the heap is initialized before being used, all values in the heap are initialized to type-dependent default values as follows:

`bool`	**`false`**
`int`	`0`
`char`	`'0'`
`string`	`""`
`T*`	`NULL`
`T[]`	An array of dimension 0
`struct` `...`	Each field initialized according to its type.

Heap values may be read by de-referencing a pointer value. Because pointers always point to the heap, there is no explicit de-allocation of heap memory (no `free`, no casting of pointer types and no pointer arithmetic in the language, a pointer value is either `NULL` or a valid address in the heap. There is also no means to obtain a pointer to a variable. The reasons for this are discussed in section 2.10.3.

monop:

`*`

lhs:

`*` *lhs*

```
int* x = alloc(int);
// Reads the value from the heap at the address stored in x.
int y = *x;
// Writes the value 9 to the heap at the address stored in x.
*x = 9;
y = 2
// y = 2, *x = 9
```

2.6 Arrays

C_0 supports variable-sized mono-typed-element arrays much like those found in Java or C#. Array access is checked against the bound of the array, yielding a runtime error if the bounds are exceeded. Arrays are passed around by reference; their values are stored in the heap.

Arrays have their element type declared as in Java: `int[] x;` This is a departure from the syntax of C but the semantics are also different. The semantics of C_0 arrays are identical (modulo bounds checking) to those of pointers of the same base type in C.

An array is created by the `alloc_array` which takes the array's element type and the array dimensions. If the dimensions are less than 0, a runtime error is issued and program execution is aborted. Arrays of dimension 0 are permitted but rather useless. All elements of the new array are initialized as described in section 2.5.

Array access uses the familiar C syntax:

expression:

expression `[` *expression* `]`

atomic-expression:

lhs `[` *expression* `]`

```
int[] x = alloc_array(int, 4);
x[0] = 9;
x[1] = 3;
x[2] = x[0] + x[1];
/* Runtime errors: */
x[4] += 9;
int[] x = alloc_array(int, -1);
```

2.7 Structures

C_0 includes structure types like those found in C - a nominally-typed set of name/value pairs (fields). Because structures allow for recursive types, there are some additional complications so that compatibility with C is maintained. Structure types may not be declared in functions - they are instead declared at the global scope along with functions.

ty:
    struct-decl
struct-decl:
    `struct` *struct-name*
struct-def:
    struct-decl { *struct-fields*$_{opt}$ }
struct-fields:
    field-decl
    struct-fields field-decl
field-decl:
    ty field-name `;`
gdecl:
    struct-decl `;`
    struct-def `;`

As with functions, C_0 makes a distinction for types between declaring and defining. A type must be defined in order to use it with a local variable declaration or structure field. The primitive intrinsic types **`void`**, **`bool`**, **`int`**, **`char`**, **`string`** are always defined. Any type that is defined is also considered to be declared. Pointer and array types are considered defined if their base type is declared.

A structure may not be defined twice but it may be declared more than once.

A structure definition may reference itself.

A type must be defined before it can be used with the `alloc` or `alloc_array` expressions. In keeping with the limitations of C, its definition must be parsed by the time the compiler parses the allocation expression.

There are no equality or assignment operators provided for structures. The motivation for their omission is to force students to think about how their mutable data structures ought to be copied and compared. Due to the lack of intrinsic support for these operations, structure types may not be used for parameters, or function return types. They are also prohibited from being used with local variables due to the lack of an equivalent to C's address-of operator coupled with aforementioned restrictions. As such, all structures are allocated on the heap and pointers to structure types are how structures are intended to be used. Though this is not promoting good practice when writing programs in C, we deemed the additional complications from allowing safe access to structures on the stack to be unacceptable.

2.8 Typedef

Since the name of a type may not always be the best description of its purpose, C_0 provides the ability to alias types by providing new names.

ty:
    type-name
type-def:
    `typedef` *ty type-name* `;`
gdecl:
    type-def

The type must be declared before it can be used with the **`typedef`** construct. Any program text after the **`typedef`** may use the alias in place of the full type.

2.9 Reasoning

Due to its simplicity and safety properties, C_0 allows the programmer to reason about their programs via annotations. These annotations are not part of the core language; they are specified via a special syntax that is embedded in the comments of the program much like JML [12], D [1] and Eiffel [5]. Annotations allow the programmer to declare pre- and post- conditions for functions, loop invariants, and general assertions. Library functions declarations can also be annotated.

These assertions about programs are checked at compile time when possible and at runtime when not. If they fail at runtime, an exception is raised and the program terminates.

Annotations are designed to encourage students to reason about their code however they can only check what the programmer has intended to check and may even contain bugs. The following example is an implementation of binary search which contains at least one bug that is not caught by the assertions:

```
int binsearch(int[] A, int n, int e)
  //@ requires \length(A) == n;
  //@ requires is_sorted(A,n);
  /*@ ensures \result >= 0
            ? A[\result] == e
            : (!contains(A, n, e) &&
               should_insert_at(A, n, -\result-1, e));
  */
  //@ ensures \result >= 1 ? A[\result-1] < A[\result] : true;
{
  int low = 0;
  int high = n-1;
  while (low <= high)
    //@ loop_invariant 0 <= low && high < n;
```

```
  //@ loop_invariant low == 0 || A[low-1] < e;
  //@ loop_invariant high == n-1 || e <= A[high+1];
{
  int mid = low + (high - low)/2;
  if (low == high)
    if (A[mid] == e)
      return mid;
    else if (A[mid] < e)
      return -1 - low;
    else
      return -low;
  else if (e <= A[mid])
    high = mid;
  else if (e > A[mid])
```

2.10 Comparison with C

C_0 is almost but not quite a subset of C. Though we strived to stay as close to C as possible, the array semantics proved impossible to resolve. However, it is still quite easy to convert a C_0 program to C such that the meaning of the C_0 program is one of the possible meanings of its C version. The conversion process requires 2 steps:

1. Convert array declarations by replacing all occurrences of `[]` with `*`.
2. Insert `#include "c0defs.h"` at the top of each

   C_0 file. `c0defs.h` defines the macros for the `alloc` and `alloc_array` expressions using `calloc` to allocate the memory and **`sizeof`** to determine the bytes. It would also need to define the **`bool`** type as well as **`true`** and **`false`** since those are not defined in C. Any library headers used would also need to be converted and included. A sample `c0defs.h` is included in the appendix.

2.10.1 Differences

Differences between C_0 and C fall into two categories: omissions from C and changes from C. There are many examples of the former:

- No unions
- No casting
- No pointer arithmetic
- No sizeof operator
- No address-of operator (&)
- No storage modifiers (static, volatile, register, extern)
- No labels, goto, switch statements or do...while loops
- No assignments in expressions

- No floating point datatypes or literals
- No complex types
- No unsigned or other integral types besides int
- Structure types may not be declared as local variables or used as return types for functions
- No comma separated expressions
- No explicit memory deallocation
- Allocation is done using types not a size in bytes.
- No fixed size arrays
- No stack allocated arrays

As previously noted, the array semantics are different in C_0 than C, matching the pointer index semantics instead.

2.10.2 Union Support

It is unfortunate that C_0 does not contain any good constructs for disjoint types. Though C has the **`union`** construct which supports them in an unsafe way, getting a type-safe version of them while maintaining the design goals of the language is rather difficult. The straightforward approach would be to layout the union in memory as in C and keep a hidden tag that is updated whenever a field in the union is written and checked when read. The major drawback to this approach is that there is no way to inspect the tag without attempting to read a value. One convention in C is to nest unions in a struct and use a separate struct field to indicate the tag. Either this separate field must be kept in sync with the hidden tag or else new syntax/semantics must be invented to tie the tag field to the union. The former is an undue burden on the programmer and the latter would deviate even more from C.

The issue is deeper however. In C and other languages, unions keep the tag and data separate which leads to issues such as the classic variant record update problem in Pascal, later encountered in Cyclone, another strongly typed imperative C-like language, [9]. The problem is that if you allow aliases to the data stored in a union, you can break type safety because users of those aliases won't know to typecheck their writes and reads. In the following example, neither function is obviously incorrect until their definitions are used together.

```
struct Box {
  // 0 for int, 1 for pointer
  int tag;
  union U {
    int i;
    int *p;
  } value;
};

int *f(Box *b, int *p) {
  b->tag = 1;
  b->value.p = p;
```

```
  return p;
}

int main () {
  Box b;
  b.tag = 0;
  b.value.i = 0;

  int *s = f(&b, &b.value.i);

  // Can now get a pointer to arbitrary memory
  *s = 40;

  return 0;
}
```

The real fix is to always tie the tag and the data together. In the ML family of languages, this is accomplished with constructors which take the appropriately typed data and add the right tag. Examining the data stored in a union requires a case analysis of the tag which allows access to the data. Even with the addition of constructors and case analysis, aliasing is still an issue in C's memory model if access to the data is given by reference. Consider the following example:

```
union Box {
  int Int;
  int* Ptr;
};

void SomeComputation(Box &b);

int main () {
  Box b = Int (0);
  Box *pb = &b;

  SomeComputation(&b);

  switch (b) {
  case Int i:
    *pb = Pointer(malloc(sizeof(int)));
    return i;
  case Pointer p:
    *pb = Int(0);
    return *p;
  }
}
```

A solution here would to copy the value and provide that to the case body instead of a reference. Cyclone supports this solution as well as forbidding assignment [8].

2.10.3 Lack of & for local variables

In C, it is a common idiom to return multiple values from a function by requiring the caller pass a pointer to a location where the callee can write them. Since the callee typically never stores the pointer anywhere, the caller can expect to safely pass the address of a stack variable instead of allocating a variable on the heap. For C_0, we'd like to permit this in a typesafe manner. We could do so by introducing an unescaping pointer type which cannot be stored into the heap. Having two types of pointers is rather strange so it would be nice to allow regular pointers to be treated as unescaping pointer but this necessitates the introduction of subtyping which so far has been avoided. It would also entail that most pointers in function declarations would be rewritten to use the unescaping pointer type instead of the regular pointer type since there is no conversion from a potentially escaping pointer to one that does not. This puts more distance between C_0 and C in a fairly intrusive way.

2.11 Analysis

C_0 strikes a balance between being too limited and too complex. Though clearly unsuitable for writing large programs, it provides sufficient syntax and semantics for expressing the data structures and algorithms used in the course. In comparison to C, it removes a number of dangerous features to gain type safety while maintaining the core syntax and semantics of the language. Though not a subset, it is close enough so that

1. A single lecture would be enough to instruct students in proper usage of C for writing the equivalent programs.
2. The meaning of a well-formed (terminates without exceptions) C_0 program is one of the possible meanings of its transliterated C version.

Chapter 3

Properties

The prior section specified the behavior of C_0 via examples and prose description. To make the specifications and expected behavior clear, we must define not only the concrete syntax, but also an abstract one whose semantics are unambiguously defined. The abstract syntax follows the concrete syntax fairly closely but omits some features from the concrete language such as strings, characters and annotations because they are not interesting or essential to the formal model.

3.1 Concrete syntax

The grammar specification below does not give definitions for the terminals *identifier*, *integer-constant*, *character-constant* and *string-constant*. These terminals are defined as follows:

identifier An identifier is a sequence of letters, digits, and underscores. The first character must be a letter. Identifiers are case sensitive.

integer-constant Integer literals come in two forms. A decimal literal is a sequence of digits (`0` through `9`) where the first digit is not 0. Hexadecimal literals begin with the two character sequence 0x and are followed by one or more hexadecimal digits (`0` through `9`, `a` through `f`). Hexadecimal literals are case insensitive.

character-constant A character constant is a single ASCII character or one of the permitted escape sequences enclosed in single quotes (`'x'`). The character `'` must be escaped. The following escape sequences are permitted:

`\t`	tab	`\b`	backspace
`\r`	return	`\n`	newline
`\'`	single quote	`\"`	double quote
`\0`	null		

string-constant A string constant (or literal) is a sequence of characters and escape sequences enclosed in double quotes (`"Hello"`). The set of permitted escape sequences for strings is the same as for characters.

bool-constant:

`true`
`false`

pointer-constant:
`NULL`

constant:
bool-constant
integer-constant
character-constant
string-constant
pointer-constant

variable-name:
identifier

function-name:
identifier

type-name:
identifier

field-name:
identifier

monop:
any of `* ! - ~`

binop:
any of `+ - * / % & | ^ && || => << >> < > <= >= == !=`

assign-op:
any of `= += -= *= /= %= &= |= ^= <<= >>=`

post-op:
any of `++ --`

expression:
constant
variable-name
expression binop expression
monop expression
expression `[` *expression* `]`
function-name
call-expression
expression `->` *field-name*
expression `.` *field-name*
expression `?` *expression* `:` *expression*
`alloc` `(` *ty* `)`
`alloc_array` `(` *ty* `,` *expression* `)`
`(` *expression* `)`

call-expression:
lhs `(` *argument-list*$_{opt}$ `)`

argument-list:
expression

argument-list `,` *expression*

atomic-expression:

variable-name

`(` `*` *lhs* `)`

lhs `[` *expression* `]`

lhs `->` *field-name*

lhs `.` *field-name*

lhs:

atomic-expression

`*` *lhs*

declaration:

ty variable-name

ty variable-name = *expression*

simple-statement:

lhs assign-op exp

call-expression$_{opt}$

atomic-expression post-op

initializer:

declaration

simple-statement

jump-statement:

`return` *expression*$_{opt}$ `;`

`break` `;`

`continue` `;`

selection-statement:

`if` `(` *expression* `)` *statement*

`if` `(` *expression* `)` *statement* `else` *statement*

iteration-statement:

`while` `(` *expression* `)` *statement*

`for` `(` *initializer* `;` *expression*$_{opt}$ `;` *simple-statement* `)` *statement*

compound-statement:

{ *statement** }

statement:

initializer `;`

jump-statement

selection-statement

iteration-statement

compound-statement

struct-decl:

`struct` *struct-name*

ty:

type-name

`void`

`bool`

`int`
ty `*`
ty `[]`
struct-decl

struct-def:
struct-decl { *struct-fields*$_{opt}$ }

struct-fields:
field-decl
struct-fields field-decl

field-decl:
ty field-name `;`

function-decl:
ty function-name `(` *function-params*$_{opt}$ `)`

function-def:
function-decl compound-statement

function-param:
ty variable-name

function-params:
function-param
function-params `,` *function-param*

type-def:
`typedef` *ty type-name* `;`

gdecl:
struct-decl `;`
struct-def `;`
type-def
function-def
function-decl `;`

program:
gdecl
program gdecl

The binary and unary operators have the following precedence and associativity in order from highest to least:

Operators	Associativity
`() [] -> .`	Left
`! ~ - *`	Right
`* / %`	Left
`+ -`	Left
`<< >>`	Left
`< <= > >=`	Left
`== !=`	Left
`&`	Left
`^`	Left
`|`	Left
`&&`	Left
`||`	Left
`=>`	Left
`? :`	Right
`= op=`	Right

There is one ambiguity in the grammar which is resolved as follows:

Matching `else` branches in nested `if` statements

As in C, in the case of

```
if (a)
if (b)
  c;
else
  d;
```

the **`else`** branch is bound to the innermost **`if`** statement.

3.1.1 Abstract syntax

We start with the abstract syntax's types τ. There are two basic types: $\mathbf{bool}$ and $\mathbf{int}$. Given a type τ we can construct pointer and array types $\tau*$ and $\tau[]$. Function types are denoted as $\tau' \to \tau$ Structure types are represented as $\texttt{struct}(p)$ where p describes the fields. p is an ordered, possibly empty, mapping of field names to types. The definition of p is written as follows:

p ::= $\cdot \mid p, f : \tau$

Two other types are used internally in the abstract syntax - they cannot be writte in the concrete language. The first is the command type $\mathbf{cmd}$ which represents the type of statements such as loops or declarations. The other is a tuple type $\texttt{tuple}(\tau_1, \ldots, \tau_n)$ which is composed of 0 or more types. These are used in the elaboration of the concrete **`void`** type and as the type for function arguments. The full definition of τ is written as follows:

$$
\begin{array}{rcl}
\tau & ::= & \mathbf{bool} \mid \mathbf{int} \mid \tau' \to \tau \\
& \mid & \tau* \\
& \mid & \tau[] \\
& \mid & \texttt{struct}(p) \\
& \mid & \mathbf{cmd} \mid \texttt{tuple}(\tau_1, \ldots, \tau_n)
\end{array}
$$

The heap is modeled as a mapping of addresses a to values or functions. The addressing model in the abstract syntax is more expressive than that of the concrete syntax. In particular it allows addresses into arrays and structure fields. A basic address is just a label that identifies some value or function in the heap. These are obtained from evaluating `alloc`. An address may also be formed by combining a label and an integer offset. These are used in intermediate expressions involving arrays such as `a[i] += 9;` where a temporary pointer needs to be created to point to the `i`th index of `a`. Similarly, structure fields can be addressed by combining an address with a field name. Structure fields addresses need to use an arbitrary address instead of a label because structures can nest inside each other and can be used as the element type of an array. Thus the definition of addresses is writen as follows:

$$
\begin{array}{rcl}
a & ::= & \mathbf{null} \\
& | & l \\
& | & l + n \\
& | & a + f
\end{array}
$$

Values are rather straightforward. $\mathbf{true}$ and $\mathbf{false}$ are inhabitants of the $\mathbf{bool}$ type. $\overline{n}$ represents an integer n. $\texttt{ptr}(a)$ is a pointer with address a with type $\tau*$ where the value at address a has type τ. Similarly, $\texttt{array}(a, n)$ is a reference to an array with base address a of length n. $\texttt{rec}(a)$ is a reference to a structure with base address a. $\texttt{func}(x_1, \ldots, x_n, e)$ represents a function with bound variables $x_1, \ldots x_n$ and function body e. Type $\mathbf{cmd}$ has one value `nop` which represents a command that does nothing. There are no values of type $\texttt{tuple}(\ldots)$ however the empty tuple $\texttt{tuple}()$ is treated as a value for the purposes of returning from functions.

$$
\begin{array}{rcl}
v & ::= & \mathbf{true} \mid \mathbf{false} \\
& | & \overline{n} \\
& | & \texttt{ptr}(a) \\
& | & \texttt{array}(a, n) \\
& | & \texttt{rec}(a) \\
& | & \texttt{func}(x_1, \ldots, x_n, e) \\
& | & \texttt{nop}
\end{array}
$$

Unlike most other languages, expressions are permitted to have both local and non-local side effects. They encompass both statements and expressions in the concrete language. The simplest expressions are variables x and values v. Binary and unary operations are represented with $\texttt{binop}(op, e_1, e_2)$ and $\texttt{monop}(op, e)$. Tuple expressions look just like their concrete counterpart. Function calls are represented as $\texttt{call}(e_f, e)$ where e_f is an expression for a pointer to the function to be called and e is the expression for the argument tuple. **`if`** statements and the conditional operator are represented by the $\texttt{if}(e_c, e_t, e_f)$ structure. Variable declarations $\texttt{decl}(x, \tau, e)$ restrict the declared variable's scope to e. $\texttt{assign}(x, e)$ represents assignments to local variables. $\texttt{return}(e)$ represents the **`return`** statement from the concrete language. $\texttt{loop}(e_c, e)$ represents a basic **`while`** loop with condition e_c and body e. **`break`** and **`continue`** are straightforward. $\texttt{alloc}(\tau)$ is an expression whose evaluation allocates a value of type τ on the heap. Similarly, $\texttt{allocarray}(\tau, e)$ is an expression whose evaluation allocates an array of type $\tau[]$ with length given by the value of e.

Noticeably missing from these expressions are memory reads/writes and sequencing of expressions. These operations are represented as binary and unary operations. The `seq` binary operator sequences two statements. The `write` binary operator takes an address and a value and

writes the value to the address in memory (or generates an exception). Similarly, the `read` unary operator takes an address and produces the value at that address (or an exception). The unary `field`(f) and binary `arrayindex` provide a means for "pointer arithmetic" so that addresses to fields of structures and elements of arrays can be produced. Finally, the `ign` unary operator allows for the evaluation of an expression for its side effects to be treated as a command.

$$
\begin{array}{rcl}
e & ::= & x \\
& | & v \\
& | & \texttt{binop}(op, e_1, e_2) \\
& | & \texttt{monop}(op, e) \\
& | & (e_1, \ldots, e_n) \\
& | & \texttt{call}(e_f, e) \\
& | & \texttt{if}(e_c, e_t, e_f) \\
& | & \texttt{decl}(x, \tau, e) \\
& | & \texttt{assign}(x, e) \\
& | & \texttt{return}(e) \\
& | & \texttt{loop}(e_c, e) \\
& | & \texttt{break} \\
& | & \texttt{continue} \\
& | & \texttt{alloc}(\tau) \\
& | & \texttt{allocarray}(\tau, e) \\
op & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{mod} \\
& | & \texttt{bitand} \mid \texttt{bitor} \mid \texttt{bitxor} \\
& | & \texttt{shl} \mid \texttt{shr} \\
& | & \texttt{cmpg} \mid \texttt{cmpl} \mid \texttt{cmpge} \mid \texttt{cmple} \mid \texttt{cmpeq} \mid \texttt{cmpne} \\
& | & \texttt{seq} \\
& | & \texttt{write} \mid \texttt{arrayindex} \\
& | & \texttt{neg} \mid \texttt{lognot} \mid \texttt{bitnot} \mid \texttt{ign} \mid \texttt{read} \\
& | & \texttt{field}(f)
\end{array}
$$

To statically check if `break` and `continue` are used in loops, this simple structure is used:

$$L ::= \texttt{inloop} \mid \texttt{notinloop}$$

When evaluating a function, it is necessary to store some information such as pending expressions and the values of local variables. This information is stored in function frames F. A function frame is an ordered list of the following: variables that are declared but not assigned ($F, x : \tau$), the values of assigned variables ($F, x : \tau_x = v$), expressions that are yet to be evaluated pending the evaluation of their subexpressions (F, e), and a construct to indicate a pending loop ($F, \texttt{loopctx}(e_c, e)$). The latter is used for the execution of the `break` and `continue` expressions.

$$
\begin{array}{rcl}
F & ::= & \cdot \mid F, e \mid F, x : \tau = v \mid F, x : \tau \\
& | & F, \texttt{loopctx}(e_c, e)
\end{array}
$$

A stack K is an ordered list of function frames:

$$K ::= \cdot \mid K; F$$

The top of the stack t is represented explicitly. It is either an expression or an exception.

$$t ::= e \mid \texttt{exn}$$

There is also a direction associated with the top to indicate if it is needs potentially further evaluation ($\triangleright$) or if it is ready to be used in evaluating the rest of the call stack ($\triangleleft$).

$\bowtie$ ::= $\triangleright \mid \triangleleft$

Evaluation involves two primary components:

- An abstract heap μ with signature Σ containing values v at addresses a.
- An abstract stack K with top t and direction $\bowtie$ yielding a value of type τ.

The initial heap contains entries for the functions and globals defined by the program and the initial stack is empty and the initial top is a call to the "main" function of the program with no arguments.

Evaluation terminates when either a value or exception is on top and the rest of the stack is empty.

3.2 Elaboration to Abstract Syntax

In order to formalize the language definition, we must provide an elaboration from the concrete syntax to the abstract syntax. Though mostly straightforward, there are some non-trivial portions of the elaboration:

Declarations In the concrete syntax, declarations extend to the end of the block. In the abstract syntax, there is no notion of "block." Instead declaration statements explicitly contain the scope of the declaration. When elaborating a declaration in a block, the rest of the block is first elaborated into some abstract statement s and then the declaration can then wrap s.

Short-circuit boolean operators The abstract syntax has no notion of the short-circuiting logical boolean operators. Both of them can be desugared into a use of the ternary operator as follows:

```
a && b  ⇒  a ? b : false
a || b  ⇒  a ? true : b
```

The ternary operator `a ? b : c` elaborates to $\texttt{if}(a, b, c)$.

Empty statements These elaborated to $\texttt{monop}(\texttt{ign}, \textbf{false})$

For-loops The **`for`** loop can be transformed into a **`while`** loop to elaborate it. The post-body statement is inserted into the loop body at the end and at every location with a non-nested **`continue`**. Call this `body'`. The loop **`for`** `(init; cond; post) body`

is rewritten as `init;` **`while`**`(cond) body'`.

Array indices `a[i]` is elaborated as $\texttt{monop}(\texttt{read}, \texttt{binop}(\texttt{arrayindex}, a, i))$ where a and i are the elaborations of `a` and `i` respectively.

Structure field projection Field projection comes in two forms. The first `e->f` elaborates to $\texttt{monop}(\texttt{read}, \texttt{monop}(\texttt{field}(f), e))$. The second `e.f` is trickier to handle since the type of e in a well-typed program should be a structure type, not a pointer type. By induction on the elaboration of `e` we know that the elaboration of `e` can be canonicalized the form $\texttt{monop}(\texttt{read}, e')$. Intuitively this makes sense since all structures are allocated on the heap. Thus, `e.f` elaborates to $\texttt{monop}(\texttt{read}, \texttt{monop}(\texttt{field}(f), e'))$.

Compound assignment For compound assignment, the elaboration depends on the left hand side.

`x op= e` This elaborates to $\texttt{assign}(x, \texttt{binop}(op, x, e))$.

`*p op= e` The left hand side elaborates to $\texttt{monop}(\texttt{read}, p)$ where p is the elaboration of `p`. Thus the elaboration becomes
$\texttt{decl}(p', \tau*, \texttt{binop}(\texttt{seq}, \texttt{assign}(p', p), \texttt{monop}(\texttt{ign}, \texttt{binop}(\texttt{write}, p', s))))$ where p' is a fresh variable and $s = \texttt{binop}(op, \texttt{monop}(\texttt{read}, p'), e)$.

Arrays and structures Since array index expressions and structure field projections elaborate to the same form as pointers, the procedure is the same as with pointers.

Functions with `void` return type The **`void`** type is elaborated to $\texttt{tuple}()$ and empty return statements are simply elaborated to $\texttt{return}(())$. Since these functions are not required to have a return statement to terminate them, one is implicitly added to the end of the function body.

Structure type elaboration Because the abstract syntax does not have nominal typing, structural types in the concrete syntax are elaborated by using the name of the structure to rename the fields of the structure. References in code to these fields must also be renamed. Structure types without any fields must remain distinct in the elaboration so a dummy field must be added to them prior to elaboration otherwise **`struct`** `F {}` and **`struct`** `G {}` would both elaborate to $\texttt{struct}(\cdot)$. Here are some examples:

Concrete	Abstract
`struct` `F {};`	$\texttt{struct}(\cdot, F\$\$dummy : \mathbf{bool})$
`struct` `G { bool dummy; }`	$\texttt{struct}(\cdot, G\$dummy : \mathbf{bool})$
`struct` `Point { int x; int y; }`	$\texttt{struct}(\cdot, Point\$x : \mathbf{int}, Point\$y : \mathbf{int})$
`struct` `Size { int x; int y; }`	$\texttt{struct}(\cdot, Size\$x : \mathbf{int}, Size\$y : \mathbf{int})$

Since \$ is not permitted in field names or structure names in C_0, it is safe to use as a delimeter.

Order of declarations for functions/types and typedefs The abstract syntax does not take the order of declarations into account. The elaborator can ignore the declarations and elaborate the definitions. Typedefs are resolved to their underlying structural type during elaboration.

3.3 Proof of safety

The proof of safety for this abstract language is given by two theorems called progress and preservation. They are defined in A.5.1 and A.5.2.

Progress

If $\mu : \Sigma$ and $\Sigma \vdash K \bowtie t : \tau$ then either $\Sigma \vdash K \lhd t$ `final` or $\mu \mid K \bowtie t \to \mu' \mid K' \bowtie' t'$ for some μ', K', $\bowtie'$, and t'.

Preservation

If $\mu : \Sigma$ and $\Sigma \vdash K \bowtie t : \tau$ and $\mu \mid K \bowtie t \to \mu' \mid K' \bowtie' t'$ then $\exists \Sigma'.\mu' : \Sigma'$ and $\Sigma' \vdash K' \bowtie t' : \tau$ and $\Sigma \leq \Sigma'$.

Chapter 4

Implementation

For 15-122, Principles of Imperative Computation, our implementation consists of a compiler and three interchangable runtimes with different characteristics. Due to the simplicity of the language and the design of the compiler, there is very little that the runtime needs to do.

4.1 Compiler

The compiler, `cc0`, performs typechecking and optional elaboration of dynamic specifications. It produces a C program which is then compiled with a runtime using `gcc`. Since the C compiler need not follow the same semantics required by C_0, `cc0` converts the program into A-normal form and uses externally defined functions for certain operations such as division and bit shifts. The optimization level for `gcc` is set very low because some of its optimizations assume C's semantics (ex: integer overflow is undefined in C) which can lead to incorrect program execution.

`cc0` uses the following representations for C_0's types

C_0 Type	C/C++ Type
`bool`	`bool`
`char`	`char`
`int`	`int32_t`
`T*`	`T*`
`T[]`	`c0_array*`
`struct S`	`struct S`
`string`	`c0_string`

To ease interoperability with native code, structures are laid out in memory using the same rules as C for member alignment and structure size. Each field of a struct is aligned according to its C/C++ type. Array elements are also aligned as in C.

Functions use the standard `cdecl` calling convention for easy interaction with C. To avoid collisions between C functions and C_0 functions, C_0 function names and references are mangled to produce a unique but fairly human-readable name. The compiler simply takes the function/-global variable name as written in C_0 and prepends `_c0_` to them. References to functions declared in libraries are not mangled.

4.2 Runtimes

We've developed three runtimes for C_0: `bare`, `c0rt`, and `unsafe`. Although our runtimes have only been tested for x86 and x86-64, the language was designed to be purposely abstract enough to be easily ported to other popular architectures.

The exact representation of arrays is provided by the runtime that the program is linked against. Access to members and the array's length is provided via runtime functions. Runtimes provide the interface declared in Appendix C.2 for use by the generated code and any libraries:

Each runtime was built to suit a particular use case.

- The `bare` runtime is an extremely basic runtime which provides bounds checking but no garbage collection. Its primary purpose was to allow C_0 programs to be compiled and run on systems which aren't supported by the garbage collector library used by the other runtimes. Since many of the assignments and test programs allocate only a little memory, the lack of garbage collection is not very noticeable. Arrays are represented as contiguous allocation as in C but with a header that includes the size of the array for bounds checking.
- The `c0rt` runtime performs bounds checking but also garbage collects allocations using the Boehm-Demers-Weiser conservative collector [3]. It uses the same representation for arrays as `bare`.
- The `unsafe` runtime garbage collects allocations using the same conservative collector but performs no bounds checking. Arrays are represented exactly as in C as a raw pointer to a contiguous block of cells. Since `cc0` compiles to C, the header for the runtime can redefine the macros used for array types and element access so that the efficient direct access to array elements is used. If `gcc`'s optimizer could be trusted, this would enable C_0 programs compiled against the `unsafe` runtime to better match C for execution speed.

Since strings are represented opaquely in the language, the runtime is free to choose any representation for strings it wants. For efficient interoperability with native code, all three runtimes represent strings as in C as an array of ASCII characters with a null terminator however other representations (ex: ropes [2]) are possible.

Chapter 5

Standard Libraries

One of the best ways to get students to learn is to get them excited about projects and motivated to complete them. These interesting projects often require supporting code such as graphics or file system access which is provided as a library to students. The goal of the standard libraries is to provide just enough support to students so that they can do these interesting projects. One nice side effect is that students are briefly exposed to different fields of computer science through these libraries and this can inform their choices of electives later on in the program. [1]

5.1 Design Principles

The standard libraries were developed at the same time as the projects for the course. This allowed us to tailor each library exactly to the requirements of the assignments. The standard libraries so far just cover just a few basic areas: images, string manipulation, and basic file/console IO. Though these are the basics for the projects in the course, they are also sufficient for students to explore projects beyond the scope of the course. By the end of the course, they will know enough C (primarily from using C_0) to write their own libraries.

Though allocations within the language are garbage collected, the native libraries upon which the standard libraries are built do not use garbage collection. As in popular modern garbage collected languages like C#, manual resource management is used for external resources.

5.2 Design

Carnegie Mellon provides several clusters of computers that run Linux for students to use so our primary platform for the initial offering of the course is these RedHat Linux-based machines. Students may have little or no experience with Linux though and will probably prefer, at least initially, to develop on their own machines. Since the compiler and runtimes work on at least two major desktop platforms (OS X, Linux) and can support the third (Windows) with some effort,

[1] The standard library is also a good place to show students how to design a good API. The C standard library has some poor choices in modularity and an unfortunate distaste for function names longer than 8 characters. Unfortunately, C_0 does not provide any additional features to aid modularity so the best we can do is adopt a prefix for the different libraries

this is feasible so long as the standard library also works on these platforms. In considering which libraries to base the standard library upon, we tried to pick only libraries that would work with minimal hassle both now and to support in the future.

There are several broad areas for which we would like to have libraries support. Though some areas such as windowing are not currently needed, we must consider which libraries to use for their implementation when considering related areas such as graphics and images. The libraries' APIs were designed to be as minimal as possible and we looked at libraries used in other introductory computer science courses such as Princeton's [15] and Processing [16].

5.2.1 Input/Output

We expect that most output in the course will be for debugging purposes and not a major focus of assignments. The console IO library (`conio`) contains just 4 functions to handle the basic task of printing data to the screen and the occasional user input. The following example demonstrates the basic usage:

```
int main() {
  string name;
  print("Hello, what is your name? ");
  name = readline();
  print("Your name is ");
  printint(string_length(name));
  println(" characters long");
  return 0;
}
```

The lack of a `printf`-like function adds to the verbosity but it is not a trivial feature to add to this language. It would require the addition of C's varargs typing to the concrete syntax and typechecker as well as the formal semantics. The type of the function would have to be inferred from its format string which require the compiler to understand the format string. This would break the abstraction boundary between the compiler and libraries. An alternative is to introduce `printf` as an intrinsic part of the language where the complicated type checking would not need to be exposed to programmers.

A simple file library was added to allow programs to read lines from a file as `string`s. File handles are one of the first manually managed resources with which students interact.

5.2.2 Data Manipulation

The `string` library contains basic routines for manipulating and inspecting strings and characters. It allows strings to be joined, split, converted to and from null-terminated character arrays, compared, generated from **`int`**s and **`bool`**s and queried for their length. It also allows **`char`**s to be compared and converted to and from their ASCII representation.

5.2.3 Images

The first assignment for 15-122 had students write code to manipulate two-dimensional images. The API allows programs to load, save and manipulate the image as a C_0 array of integers representing the pixel data packed as ARGB. Like file handles, images also require manual resource management.

5.2.4 Windowing

The projects in 15-122 should require only the most basic windowing support. It defines a window as an abstraction of a keyboard, a mouse, and a screen - it receives input via mouse and keyboard and displays an image. There are no child widgets for buttons or editboxes - students can write those themselves if desired but the text console ought to be sufficient for functionality required the projects in the course. Furthermore, the text interface is much easier to write automated tests for than a graphical one.

5.2.5 Graphics

For graphics support, we want to expose a minimal but powerful API. For inspiration, we looked at Java's AWT graphics objects, HTML's `<canvas>` 2D context API, and Cairo's API. Students are able to draw and manipulate images as well as draw shapes at arbitrary positions and scales and rotations. Students are able to construct 2D geometric paths and perform basic queries on them. They can also construct transformations from primitives like translation, rotation and scale. Paths and transforms are used with a graphics context which is obtained from an image or a window. With a graphics context, basic operations such as drawing/filling a path with a transformation and clipping region allow great flexibility with a minimal API.

5.3 Implementation

Defining libraries is a rather simple process. A library named `foo` must provide a header file `foo.h0` containing the C_0 types and function declarations for the library and a shared library with the same name base name as the library (ex: `libfoo.so`, `libfoo.dylib`, `foo.dll`). The compiler is instructed to use the library `foo` by passing `-lfoo`. It will process the declarations and definitions in the library's header and link the executable against the shared library.

5.3.1 Native Dependencies

Many of the basic libraries can be implemented using the standard C library since it is available on all three major desktop platforms.

For windowing there are both many and few choices. Each platform has its own native libraries for basic window operations but supporting separate branches that work well on moving platforms such as OS X or diverse platforms such as Linux is not a good long-term approach so

we decided to look at cross-platform abstractions. There are many such libraries but the popular ones are Mozilla's XUL/Gecko library, GTK, wxWidgets, FLTK, Qt.

The size of XUL/Gecko both in terms of size and scope is simply too great for our needs. Likewise, GTK requires quite a bit of overhead on non-Linux platforms (particularly Windows) and does not integrate with Microsoft's compiler and toolchain, something that students may want to do when learning C on Windows. wxWidgets has a fairly full featured set of tools that go beyond just windowing. It does not abstract the details of the underlying platform that well so students may see different behavior than their grader and likewise the graders may not be able to see problems that the students are having. FLTK is a minimal windowing toolkit that does not attempt to look native. It provides basic graphics support as well. Like wxWidgets, Qt provides quite a bit of non-GUI support and though it doesn't have a native look and feel, it comes quite close.

The choices of graphics library is closely related to that of the windowing library. FLTK, wxWidgets and Qt provide their own graphics libraries. Out of these three, only wxWidgets does not abstract out the details of the underlying platform since wxWidgets relies upon the native platform for much of its drawing. There are also dedicated graphics libraries such as Skia and Cairo which could potentially work with whatever windowing library was chosen.

Though there are many individual libraries that would be suitable for the implementation of the major components of the standard library, Qt has the advantage of providing APIs for all the major requirements. Moreover, the interaction between these separate components is already written and tested. Qt is used by a number of major corporations and projects around the world such as Adobe[2], KDE, Mathematica, Opera and VLC. It is thus expected that Qt will remain a stable and supported platform upon which the standard library can be used by students with a minimal number of bugs and support required by the course staff.

[2]http://qt.nokia.com/qt-in-use/story/app/adobe-photoshop-album

Chapter 6

Related and Future Work

6.1 Related work

Projects like SAFECode [4] and CCured [14] are also attempts to produce safe C-like imperative languages. Unlike C_0, they are aimed more at verifying or porting existing C programs. Another attempt at an imperative systems language is Google's Go Language [6] which breaks away from C but it does contain additional features which are only useful for writing larger programs than those in 15-122. Go also does not have assertions or exception built into the language, preferring instead to return error codes and propagate or handle them explicitly without assistance from the compiler. The designers' goal is to force programmers to think about proper error handling whereas the intended focus of 15-122 is to encourage students to reason about the correctness of their code.

Processing.org [16] is another take on providing an easy and simple programming environment for novice programmers. It specializes in programming graphical applications such as games or visualizations. Though it simplifies away some of its Java roots by eliminating the need for a driver class, it is by its own admission "just Java, but with a new graphics and utility API along with some simplifications."

6.2 Future work and speculation

The work presented in this thesis is intended to be only the start of a larger project. Due to the time constraints of needing the language and compiler to be finished in time for the course in the Fall of 2010, several interesting projects and ideas have been given little attention. These include a static prover for assertions, a module system for the language to replace the one found in C, and using a precise garbage collector in the runtime as well as other more advanced garbage collection techniques.

6.2.1 Static prover for assertions

C_0 is likely simple enough to do some strong analysis of programs to prove invariants and assertions will hold at runtime. The lack of an address-of and cast operators limits aliasing and

the local variables can be hoisted to the heap by rewriting the program without changing any discernible semantics. As a project for a graduate program analysis class, Karl Naden, Anvesh Komuravelli and Sven Stork built a small static verification tool for programs. They annotated programs with assertions and invariants using JML-style notation. The work at the end of the course showed promise towards being to be able to prove some of the invariants relevant to the data structures used in the course.

6.2.2 Module system & Polymorphism

C's module system has well known faults and both C and C_0 lack a decent means of supporting polymorphism. I did some preliminary work towards defining a module system that was simple enough to be used by the course's students but powerful enough to allow for abstract polymorphism. The intention was to allow students to write generic data structures. Combined with the aforementioned static verification tool, students would be able to write assertions and independently check that their generic data structure or algorithm was correct.

6.2.3 Precise collector

Though the conservative collector in the runtime is adequate for most purposes of the course, it is not ideal. Since there is no casting or internal aliasing, it's possible to construct a more precise collector that can analyze the stack layout and walk the heap efficiently. Using a precise collector would also necessitate a change to the runtime interface to allow the garbage collector to walk library-defined types and provide a rooting API.

6.2.4 Dynamically check explicit deallocation

Tools like Valgrind are often used on C and C++ programs to check for common mistakes such as double frees and use-after-free bugs. C_0 could be augmented with the ability to explicitly free memory and at runtime (if not statically) assert that the program does not use memory after freeing it or free memory twice. This would bring the language closer to C without compromising on safety.

6.2.5 Convenient `printf` support

One of the notable features absent from C in C_0 is support for variadic functions like `printf`. Supporting these functions while maintaining type safety is rather non-trivial. Attempt to support it in languages like SML have been tried by reformulating the format string as higher order functions but that approach does not suit C_0.

6.2.6 Formatting/style checker

Good code hygiene is an often overlooked aspect of computer science education. Though there is no single style for writing good C code, there are several popular styles and techniques for all languages. Some upper level courses such as Operating Systems include a code review as

part of the grading mechanism. Though quite time consuming, it provides generally helpful feedback to students in ways that an automatic grader usually does not. Some languages such as Go [6] include a formatting program which automatically rewrites source files to a pre-defined style. The former is time consuming and prone to divergent styles/feedback and the latter does not encourage/require students to learn proper styling. A program which could consistently verify/grade the style of student code and generate useful human readable feedback, would be of enormous value to both the students and the course staff.

6.2.7 Interactive debugger

Asking students to run `gdb` on the compiled binaries is asking too much of them. We can produce a debugger for C_0 programs that can provide a friendlier interface than GDB and enable additional features such as reversible debugging. Some work on this was started before the course by Rob Simmons.

6.2.8 Feedback on the course and its use of C_0

At the time of this publication, it is too early to know what impact choice of C_0 will have. Getting feedback from the students over the course of their time at Carnegie Mellon will provide valuable data. Already from the first few weeks, suggestions for incremental changes to the language and compiler have been made from both students and staff.

Appendix A

Language Definition

A.1 Syntax

Struct fields p	::=	$\cdot \mid p, f : \tau$
Types τ	::=	$\mathbf{bool} \mid \mathbf{int} \mid \tau' \to \tau$
	$\mid$	$\tau*$
	$\mid$	$\tau[]$
	$\mid$	$\texttt{struct}(p)$
	$\mid$	$\mathbf{cmd} \mid \texttt{tuple}(\tau_1, \ldots, \tau_n)$
Address a	::=	$\mathbf{null}$
	$\mid$	l
	$\mid$	$l + n$
	$\mid$	$a + f$
Values v	::=	$\mathbf{true} \mid \mathbf{false}$
	$\mid$	$\overline{n}$
	$\mid$	$\texttt{ptr}(a)$
	$\mid$	$\texttt{array}(a, n)$
	$\mid$	$\texttt{rec}(a)$
	$\mid$	$\texttt{func}(x_1, \ldots, x_n, e)$
	$\mid$	$\texttt{nop}$
Operators op	::=	$\texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{mod}$
	$\mid$	$\texttt{bitand} \mid \texttt{bitor} \mid \texttt{bitxor}$
	$\mid$	$\texttt{shl} \mid \texttt{shr}$
	$\mid$	$\texttt{cmpg} \mid \texttt{cmpl} \mid \texttt{cmpge} \mid \texttt{cmple} \mid \texttt{cmpeq} \mid \texttt{cmpne}$
	$\mid$	$\texttt{seq}$
	$\mid$	$\texttt{write} \mid \texttt{arrayindex}$
	$\mid$	$\texttt{neg} \mid \texttt{lognot} \mid \texttt{bitnot} \mid \texttt{ign} \mid \texttt{read}$
	$\mid$	$\texttt{field}(f)$

Expression e	::=	x
	|	v
	|	$\texttt{binop}(op, e_1, e_2)$
	|	$\texttt{monop}(op, e)$
	|	$(e_1, \ldots, e_n)$
	|	$\texttt{call}(e_f, e)$
	|	$\texttt{if}(e_c, e_t, e_f)$
	|	$\texttt{decl}(x, \tau, e)$
	|	$\texttt{assign}(x, e)$
	|	$\texttt{return}(e)$
	|	$\texttt{loop}(e_c, e)$
	|	$\texttt{break}$
	|	$\texttt{continue}$
	|	$\texttt{alloc}(\tau)$
	|	$\texttt{allocarray}(\tau, e)$
Loop states L	::=	$\texttt{inloop} \mid \texttt{notinloop}$
Call frames F	::=	$\cdot \mid F, e \mid F, x : \tau = v \mid F, x : \tau$
	|	$F, \texttt{loopctx}(e_c, e)$
Stack frames K	::=	$\cdot \mid K; F$
Directions $\bowtie$	::=	$\rhd \mid \lhd$
Foci t	::=	$e \mid \texttt{exn}$
Variable contexts Γ	::=	$\cdot \mid \Gamma, x : \tau$
Memory signatures $\Sigma(a)$	=	τ
Memory stores $\mu(a)$	=	v

First we begin with the judgment $\tau\ \texttt{small}$ which determines if a type is permitted to be used as the type of a declaration, parameter of a function, or return type of a function. It specifically excludes structures, functions, non-degenerate tuples and commands.

$$\Sigma \vdash v : \tau$$

Given a value and a memory signature, this judgment determines a type for the value. Booleans and integers can be typed without the memory signature but the other types require it so that they do not need to embed any types. The null pointer is allowed to be any pointer type.

$$\Sigma; \Gamma \vdash e : \tau$$

This judgment determines the type of an expression from a context which describes the variables that are in scope and a signature of the memory. Checking that an expression only uses assigned variables is checked via a separate judgment.

$$\tau\ \texttt{alloc}$$

We must restrict the types which can be allocated to those which are directly expressible in the concrete language. This excludes tuples, commands and functions.

$$\Gamma \vdash e : A \to A'$$

Since C_0 requires variables to be assigned before being used, we use this judgment to determine the set of variables an expression requires to be assigned (A) and given that set, which set of variables will be defined after evaluating the expression (A') in a context of declared variables Γ.

$$L \vdash e \texttt{ ok}$$

This judgment captures the need to restrict expressions such as break and continue to only occur within a loop.

$$\Sigma; \Gamma \vdash e \texttt{ canreturn } \tau \qquad \vdash e \texttt{ returns} \qquad \Sigma; \Gamma \vdash e \texttt{ returns } \tau$$

Determining that a function matches its declared return type requires three judgments. The first checks that every possible execution path will, if it returns a value, return a value of the correct type. The second judgment ensures that every possible execution path which reaches the end of the function will return. The third judgment combines the first two and requires that the return type be a small type to be consistent with C_0.

$$op : \tau' \times \tau'' \to \tau \qquad op : \tau' \to \tau$$

These two judgments determine the types of the binary and unary operators in the language.

$$\Sigma; A \vdash F : \tau \to \tau'$$

Given a memory signature and a set of assigned variables, we want to know if a (partial) call frame is well formed. This judgment captures the type that the top framelet expects and the type that the call frame will eventually returns.

$$\Sigma \vdash \texttt{ctx}(F) = \Gamma \qquad \texttt{assigned}(F) = A \qquad \texttt{loopnest}(F) = L \qquad \texttt{innerloop}(F) = F'$$

These four judgments define recursive functions which extract information embedded in the stack. The first determines the typing context of the frame. The second determines which variables have been assigned. The third determines if the current call frame is in the middle of a loop or not. If the call frame is in a loop, the fourth function will return the partial stack frame which indicates the innermost loop.

$$\Sigma; \Gamma \vdash e \texttt{ pending } \tau \to \tau' \mathbin{@} \tau''$$

Expressions on the stack are always waiting for some value so that they can continue with their evaluation. This judgment captures the type of the expected value (τ), the type that the expression will return once evaluation is complete (τ'), and the return type which the expression will return, if it does return (τ''). Since some expressions have more than one subexpression, a hole ($\cdot$) is used to mark which subexpression is pending evaluation. The hole only marks an immediate subexpression of e.

$$\bowtie \; e \; \texttt{ok}$$

This judgment simply checks if an expression is permitted to be returned to the topmost framelet of the top call frame. Only values and empty tuples are permitted to be returned.

$$\Sigma \vdash K : \tau \to \tau'$$

We also must check the rest of the stack. Given a value of type τ, evaluation of the call frames will yield a value of type τ'.

$$\Sigma \vdash K \bowtie t : \tau$$

Typing all these judgments together, we can check that the entire stack K and the focus t are well formed, yielding a final value of type τ. The focus of a stack is an expression (often a subexpression of a function) that is being evalutated, a value that is being used to evaluate a pending expression on the topmost call frame, or an exception which is propagating up the stack.

$$\Sigma \vdash K \lhd t \; \texttt{final}$$

Evaluation is finished when the stack is in a final state. There are no call frames and the focus t is either an exception or a value.

$$\mu \mid op(v_1, v_2) \to \mu' \mid t \qquad\qquad \mu \mid op(v) \to t$$

These two judgments define the behavior of the binary and unary operators respectively. Binary operators such as `write` may modify memory and unary operators such as `read` may depend on memory but do not modify it. Evaluating an operator is expected to result in a value of the appropriate type but may result in an exception (ex: trying to write a value to **null**).

$$\texttt{allocval}(\mu, a, \tau) = \mu' \qquad\qquad \texttt{makearray}(\mu, a, \tau, n) = \mu'$$

These two judgments define functions which perform the allocation of values. Given a heap, an address not in the heap and a type (and a length for arrays), these functions return a new memory containing the appropriate initialized allocations.

$$\mu \mid K; F \bowtie t \to \mu' \mid K' \bowtie' t'$$

This judgment defines the transitions between well-typed program states. Evaluation consists of iterating these transitions until a final state is reached.

$$\mu : \Sigma \qquad\qquad \Sigma \leq \Sigma'$$

The first of these judgments checks that a memory matches the given signature. The second defines a partial relation between signatures which is used to check that memory operations do not destroy or alter any existing allocations.

A.2 Rules

$\boxed{\tau \texttt{ small}}$

$$\frac{}{\textbf{bool}\ \texttt{small}}\ \text{TAUSMALL-BOOL} \qquad \frac{}{\textbf{int}\ \texttt{small}}\ \text{TAUSMALL-INT} \qquad \frac{}{\tau * \texttt{ small}}\ \text{TAUSMALL-PTR}$$

$$\frac{}{\tau[]\ \texttt{small}}\ \text{TAUSMALL-ARRAY}$$

$\boxed{\Sigma \vdash v : \tau}$

$$\frac{}{\Sigma \vdash \texttt{nop} : \textbf{cmd}}\ \text{CHECKVAL-NOP} \qquad \frac{}{\Sigma \vdash \textbf{true} : \textbf{bool}}\ \text{CHECKVAL-TRUE}$$

$$\frac{}{\Sigma \vdash \textbf{false} : \textbf{bool}}\ \text{CHECKVAL-FALSE} \qquad \frac{n < 2^{31} \qquad n \geq -2^{31}}{\Sigma \vdash \overline{n} : \textbf{int}}\ \text{CHECKVAL-INT}$$

$$\frac{\Sigma(a) = \tau}{\Sigma \vdash \texttt{ptr}(a) : \tau*}\ \text{CHECKVAL-ADDRESS} \qquad \frac{}{\Sigma \vdash \texttt{ptr}(\textbf{null}) : \tau*}\ \text{CHECKVAL-NULL}$$

$$\frac{\forall i \in [0, n).\Sigma(a + i) = \tau}{\Sigma \vdash \texttt{array}(a, n) : \tau[]}\ \text{CHECKVAL-ARRAY} \qquad \frac{\forall f : \tau \in p.\Sigma(a + f) = \tau}{\Sigma \vdash \texttt{rec}(a) : \texttt{struct}(p)}\ \text{CHECKVAL-STRUCT}$$

$$\frac{\begin{array}{c}\Gamma = \cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \qquad A = \{x_1, \ldots, x_n\} \\ |A| = n \qquad \Sigma; \Gamma \vdash e : \textbf{cmd} \qquad \Gamma \vdash e : A \to A \qquad \texttt{notinloop} \vdash e\ \texttt{ok} \\ \Sigma; \Gamma \vdash e\ \texttt{returns}\ \tau \qquad \tau\ \texttt{small} \qquad \tau_1\ \texttt{small} \qquad \ldots \qquad \tau_n\ \texttt{small}\end{array}}{\Sigma \vdash \texttt{func}(x_1, \ldots, x_n, e) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau}\ \text{CHECKVAL-FUNC}$$

$\boxed{\Sigma; \Gamma \vdash e : \tau}$

$$\frac{}{\Sigma; \Gamma, x : \tau \vdash x : \tau}\ \textsc{checkexp-var} \qquad \frac{\Sigma \vdash v : \tau}{\Sigma; \Gamma \vdash v : \tau}\ \textsc{checkexp-value}$$

$$\frac{op : \tau' \times \tau'' \to \tau \qquad \Sigma; \Gamma \vdash e_1 : \tau' \qquad \Sigma; \Gamma \vdash e_2 : \tau''}{\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2) : \tau}\ \textsc{checkexp-binop}$$

$$\frac{op : \tau' \to \tau \qquad \Sigma; \Gamma \vdash e : \tau'}{\Sigma; \Gamma \vdash \texttt{monop}(op, e) : \tau}\ \textsc{checkexp-monop}$$

$$\frac{\begin{array}{ccccccc} & & & \Sigma; \Gamma \vdash e_1 : \tau_1 & & & \\ \ldots & \Sigma; \Gamma \vdash e_n : \tau_n & & \tau_1\ \texttt{small} & \ldots & \tau_n\ \texttt{small} & \end{array}}{\Sigma; \Gamma \vdash (e_1, \ldots, e_n) : \texttt{tuple}(\tau_1, \ldots, \tau_n)}\ \textsc{checkexp-tuple}$$

$$\frac{\Sigma; \Gamma \vdash e_f : (\texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau)* \qquad \Sigma; \Gamma \vdash e : \texttt{tuple}(\tau_1, \ldots, \tau_n)}{\Sigma; \Gamma \vdash \texttt{call}(e_f, e) : \tau}\ \textsc{checkexp-call}$$

$$\frac{\Sigma; \Gamma \vdash e_c : \textbf{bool} \qquad \Sigma; \Gamma \vdash e_t : \tau \qquad \Sigma; \Gamma \vdash e_f : \tau}{\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f) : \tau}\ \textsc{checkexp-if}$$

$$\frac{\Sigma; \Gamma, x : \tau \vdash e : \textbf{cmd} \qquad \tau\ \texttt{small}}{\Sigma; \Gamma \vdash \texttt{decl}(x, \tau, e) : \textbf{cmd}}\ \textsc{checkexp-decl}$$

$$\frac{\Sigma; \Gamma, x : \tau \vdash e : \tau \qquad \tau\ \texttt{small}}{\Sigma; \Gamma, x : \tau \vdash \texttt{assign}(x, e) : \textbf{cmd}}\ \textsc{checkexp-assign}$$

$$\frac{\Sigma; \Gamma \vdash e : \tau \qquad \tau\ \texttt{small}}{\Sigma; \Gamma \vdash \texttt{return}(e) : \textbf{cmd}}\ \textsc{checkexp-return}$$

$$\frac{\Sigma; \Gamma \vdash e_c : \textbf{bool} \qquad \Sigma; \Gamma \vdash e : \textbf{cmd}}{\Sigma; \Gamma \vdash \texttt{loop}(e_c, e) : \textbf{cmd}}\ \textsc{checkexp-loop}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{break} : \textbf{cmd}}\ \textsc{checkexp-break} \qquad \frac{}{\Sigma; \Gamma \vdash \texttt{continue} : \textbf{cmd}}\ \textsc{checkexp-continue}$$

$$\frac{\tau\ \texttt{alloc}}{\Sigma; \Gamma \vdash \texttt{alloc}(\tau) : \tau*}\ \textsc{checkexp-alloc}$$

$$\frac{\Sigma; \Gamma \vdash e : \textbf{int} \qquad \tau\ \texttt{alloc}}{\Sigma; \Gamma \vdash \texttt{allocarray}(\tau, e) : \tau[]}\ \textsc{checkexp-allocarray}$$

$\boxed{\tau\ \texttt{alloc}}$

$$\frac{}{\textbf{bool}\ \texttt{alloc}}\ \textsc{canalloc-bool} \qquad \frac{}{\textbf{int}\ \texttt{alloc}}\ \textsc{canalloc-int} \qquad \frac{}{\tau * \texttt{alloc}}\ \textsc{canalloc-ptr}$$

$$\frac{}{\tau[]\ \texttt{alloc}}\ \textsc{canalloc-array} \qquad \frac{}{\texttt{struct}(p)\ \texttt{alloc}}\ \textsc{canalloc-struct}$$

$\boxed{\Gamma \vdash e : A \to A'}$

$$\frac{x \in A}{\Gamma \vdash x : A \to A}\ \textsc{checkassign-var} \qquad \frac{}{\Gamma \vdash \cdot : A \to A}\ \textsc{checkassign-hole}$$

$$\frac{}{\Gamma \vdash v : A \to A}\ \textsc{checkassign-value}$$

$$\frac{\Gamma \vdash e_1 : A \to A' \qquad \Gamma \vdash e_2 : A' \to A''}{\Gamma \vdash \texttt{binop}(op, e_1, e_2) : A \to A''}\ \textsc{checkassign-binop}$$

$$\frac{\Gamma \vdash e : A \to A}{\Gamma \vdash \texttt{monop}(op, e) : A \to A}\ \textsc{checkassign-monop}$$

$$\frac{\Gamma \vdash e_1 : A \to A \qquad \ldots \qquad \Gamma \vdash e_n : A \to A}{\Gamma \vdash (e_1, \ldots, e_n) : A \to A}\ \textsc{checkassign-tuple}$$

$$\frac{}{\Gamma \vdash \texttt{call}(e_f, e) : A \to A}\ \textsc{checkassign-call}$$

$$\frac{\Gamma \vdash e_c : A \to A \qquad \Gamma \vdash e_t : A \to A' \qquad \Gamma \vdash e_f : A \to A''}{\Gamma \vdash \texttt{if}(e_c, e_t, e_f) : A \to A' \cap A''}\ \textsc{checkassign-if}$$

$$\frac{x \notin A \qquad \Gamma, x : \tau \vdash e : A \to A'}{\Gamma \vdash \texttt{decl}(x, \tau, e) : A \to A' - \{x\}}\ \textsc{checkassign-decl}$$

$$\frac{\Gamma \vdash e : A \to A}{\Gamma \vdash \texttt{assign}(x, e) : A \to A \cup \{x\}}\ \textsc{checkassign-assign}$$

$$\frac{\Gamma \vdash e : A \to A}{\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{return}(e) : A \to \{x_1, \ldots, x_n\}}\ \textsc{checkassign-return}$$

$$\frac{\Gamma \vdash e_c : A \to A \qquad \Gamma \vdash e : A \to A'}{\Gamma \vdash \texttt{loop}(e_c, e) : A \to A}\ \textsc{checkassign-loop}$$

$$\frac{}{\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{break} : A \to \{x_1, \ldots, x_n\}}\ \textsc{checkassign-break}$$

$$\frac{}{\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{continue} : A \to \{x_1, \ldots, x_n\}}\ \textsc{checkassign-continue}$$

$$\frac{}{\Gamma \vdash \texttt{nop} : A \to A}\ \textsc{checkassign-nop} \qquad \frac{}{\Gamma \vdash \texttt{alloc}(\tau) : A \to A}\ \textsc{checkassign-alloc}$$

$$\frac{\Gamma \vdash e : A \to A}{\Gamma \vdash \texttt{allocarray}(\tau, e) : A \to A}\ \textsc{checkassign-allocarray}$$

$\boxed{L \vdash e\ \texttt{ok}}$

$$\frac{}{L \vdash x\ \texttt{ok}}\ \textsc{checkloop-var} \qquad \frac{}{L \vdash v\ \texttt{ok}}\ \textsc{checkloop-value}$$

$$\frac{L \vdash e_1\ \texttt{ok} \qquad L \vdash e_2\ \texttt{ok}}{L \vdash \texttt{binop}(op, e_1, e_2)\ \texttt{ok}}\ \textsc{checkloop-binop} \qquad \frac{L \vdash e\ \texttt{ok}}{L \vdash \texttt{monop}(op, e)\ \texttt{ok}}\ \textsc{checkloop-monop}$$

$$\frac{L \vdash e_1\ \texttt{ok} \qquad \ldots \qquad L \vdash e_n\ \texttt{ok}}{L \vdash (e_1, \ldots, e_n)\ \texttt{ok}}\ \textsc{checkloop-tuple}$$

$$\frac{L \vdash e_f\ \texttt{ok} \qquad L \vdash e_a\ \texttt{ok}}{L \vdash \texttt{call}(e_f, e_a)\ \texttt{ok}}\ \textsc{checkloop-call}$$

$$\frac{L \vdash e_c\ \texttt{ok} \qquad L \vdash e_t\ \texttt{ok} \qquad L \vdash e_f\ \texttt{ok}}{L \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{ok}}\ \textsc{checkloop-if}$$

$$\frac{L \vdash e\ \texttt{ok}}{L \vdash \texttt{decl}(x, \tau, e)\ \texttt{ok}}\ \textsc{checkloop-decl} \qquad \frac{L \vdash e\ \texttt{ok}}{L \vdash \texttt{assign}(x, e)\ \texttt{ok}}\ \textsc{checkloop-assign}$$

$$\frac{L \vdash e\ \texttt{ok}}{L \vdash \texttt{return}(e)\ \texttt{ok}}\ \textsc{checkloop-return}$$

$$\frac{L \vdash e_c\ \texttt{ok} \qquad \texttt{inloop} \vdash e\ \texttt{ok}}{L \vdash \texttt{loop}(e_c, e)\ \texttt{ok}}\ \textsc{checkloop-loop}$$

$$\frac{}{\texttt{inloop} \vdash \texttt{break}\ \texttt{ok}}\ \textsc{checkloop-break}$$

$$\frac{}{\texttt{inloop} \vdash \texttt{continue}\ \texttt{ok}}\ \textsc{checkloop-continue} \qquad \frac{}{L \vdash \texttt{nop}\ \texttt{ok}}\ \textsc{checkloop-nop}$$

$$\frac{}{L \vdash \texttt{alloc}(\tau)\ \texttt{ok}}\ \textsc{checkloop-alloc}$$

$$\frac{L \vdash e\ \texttt{ok}}{L \vdash \texttt{allocarray}(\tau, e)\ \texttt{ok}}\ \textsc{checkloop-allocarray}$$

$\boxed{\Sigma; \Gamma \vdash e\ \texttt{canreturn}\ \tau}$

$$\frac{\Sigma; \Gamma \vdash e_1\ \texttt{canreturn}\ \tau \qquad \Sigma; \Gamma \vdash e_2\ \texttt{canreturn}\ \tau}{\Sigma; \Gamma \vdash \texttt{binop}(\texttt{seq}, e_1, e_2)\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-binop}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{monop}(\texttt{ign}, e)\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-monop}$$

$$\frac{\Sigma; \Gamma \vdash e_t\ \texttt{canreturn}\ \tau \qquad \Sigma; \Gamma \vdash e_f\ \texttt{canreturn}\ \tau}{\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-if}$$

$$\frac{\Sigma; \Gamma, x : \tau \vdash e\ \texttt{canreturn}\ \tau}{\Sigma; \Gamma \vdash \texttt{decl}(x, \tau, e)\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-decl}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{assign}(x, e)\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-assign}$$

$$\frac{\Sigma; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \texttt{return}(e)\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-return}$$

$$\frac{\Sigma; \Gamma \vdash e\ \texttt{canreturn}\ \tau}{\Sigma; \Gamma \vdash \texttt{loop}(e_c, e)\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-loop}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{break}\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-break}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{continue}\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-continue}$$

$$\frac{}{\Sigma; \Gamma \vdash \texttt{nop}\ \texttt{canreturn}\ \tau}\ \textsc{onlyreturns-nop}$$

$\boxed{\vdash e\ \texttt{returns}}$

$$\frac{\vdash e_1\ \texttt{returns}}{\vdash \texttt{binop}(\texttt{seq}, e_1, e_2)\ \texttt{returns}}\ \textsc{doesreturn-binoplhs}$$

$$\frac{\vdash e_2\ \texttt{returns}}{\vdash \texttt{binop}(\texttt{seq}, e_1, e_2)\ \texttt{returns}}\ \textsc{doesreturn-binoprhs}$$

$$\frac{\vdash e_t \texttt{ returns} \qquad \vdash e_f \texttt{ returns}}{\vdash \texttt{if}(e_c, e_t, e_f) \texttt{ returns}} \text{ DOESRETURN-IF}$$

$$\frac{\vdash e \texttt{ returns}}{\vdash \texttt{decl}(x, \tau, e) \texttt{ returns}} \text{ DOESRETURN-DECL}$$

$$\frac{}{\vdash \texttt{return}(e) \texttt{ returns}} \text{ DOESRETURN-RETURN}$$

$\boxed{\Sigma; \Gamma \vdash e \texttt{ returns } \tau}$

$$\frac{\Sigma; \Gamma \vdash e \texttt{ canreturn } \tau \qquad \vdash e \texttt{ returns} \qquad \tau \texttt{ small}}{\Sigma; \Gamma \vdash e \texttt{ returns } \tau} \text{ RETURNS-ONLY}$$

$\boxed{op : \tau' \times \tau'' \to \tau}$

$$\frac{}{\texttt{add} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPADDINTEGER}$$

$$\frac{}{\texttt{sub} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPSUBINTEGER}$$

$$\frac{}{\texttt{mul} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPMULINTEGER}$$

$$\frac{}{\texttt{div} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPDIVINTEGER}$$

$$\frac{}{\texttt{mod} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPMODINTEGER}$$

$$\frac{}{\texttt{bitand} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPBITANDINTEGER}$$

$$\frac{}{\texttt{bitor} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPBITORINTEGER}$$

$$\frac{}{\texttt{bitxor} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPBITXORINTEGER}$$

$$\frac{}{\texttt{shl} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPSHLINTEGER}$$

$$\frac{}{\texttt{shr} : \mathbf{int} \times \mathbf{int} \to \mathbf{int}} \text{ CHECKBINOP-OPSHRINTEGER}$$

$$\frac{}{\texttt{cmpg} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPGINTEGER}$$

$$\frac{}{\texttt{cmpl} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPLINTEGER}$$

$$\frac{}{\texttt{cmpge} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPGEINTEGER}$$

$$\frac{}{\texttt{cmple} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPLEINTEGER}$$

$$\frac{}{\texttt{cmpeq} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPEQINTEGER}$$

$$\frac{}{\texttt{cmpne} : \mathbf{int} \times \mathbf{int} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPNEINTEGER}$$

$$\frac{}{\texttt{cmpeq} : \mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPEQBOOL}$$

$$\frac{}{\texttt{cmpne} : \mathbf{bool} \times \mathbf{bool} \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPNEBOOL}$$

$$\frac{}{\texttt{cmpeq} : \tau* \times \tau* \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPEQPOINTERTAU}$$

$$\frac{}{\texttt{cmpne} : \tau* \times \tau* \to \mathbf{bool}}\ \text{CHECKBINOP-OPCMPNEPOINTERTAU}$$

$$\frac{}{\texttt{seq} : \mathbf{cmd} \times \mathbf{cmd} \to \mathbf{cmd}}\ \text{CHECKBINOP-SEQ} \qquad \frac{\tau\ \texttt{small}}{\texttt{write} : \tau* \times \tau \to \tau}\ \text{CHECKBINOP-WRITE}$$

$$\frac{}{\texttt{arrayindex} : \tau[] \times \mathbf{int} \to \tau*}\ \text{CHECKBINOP-ARRAYINDEX}$$

$\boxed{op : \tau' \to \tau}$

$$\frac{}{\texttt{neg} : \mathbf{int} \to \mathbf{int}}\ \text{CHECKMONOP-OPNEGINTEGER}$$

$$\frac{}{\texttt{lognot} : \mathbf{bool} \to \mathbf{bool}}\ \text{CHECKMONOP-OPLOGNOTBOOL}$$

$$\frac{}{\texttt{bitnot} : \mathbf{int} \to \mathbf{int}}\ \text{CHECKMONOP-OPBITNOTINTEGER} \qquad \frac{\tau\ \texttt{small}}{\texttt{ign} : \tau \to \mathbf{cmd}}\ \text{CHECKMONOP-IGN}$$

$$\frac{\tau\ \texttt{small}}{\texttt{read} : \tau* \to \tau}\ \text{CHECKMONOP-READ}$$

$$\frac{1 \leq i \leq n}{\texttt{field}(x_i) : \texttt{struct}(\cdot, f_1 : \tau_1, \ldots, f_n : \tau_n)* \to \tau_i*} \text{ CHECKMONOP-FIELD}$$

$\boxed{\Sigma; A \vdash F : \tau \to \tau'}$

$$\frac{\begin{array}{c}\Sigma; A' \vdash F : \tau'' \to \tau' \qquad \Gamma \vdash e : A \to A' \\ \Sigma \vdash \texttt{ctx}(F) = \Gamma \qquad \texttt{loopnest}(F) = L \\ L \vdash e\ \texttt{ok} \qquad \Sigma; \Gamma \vdash e\ \texttt{pending}\ \tau \to \tau'' \ @\ \tau'\end{array}}{\Sigma; A \vdash F, e : \tau \to \tau'} \text{ CHECKFRAMETYPE-FRAMEEXPNORET}$$

$$\frac{\begin{array}{c}\vdash e\ \texttt{returns} \qquad \Sigma \vdash \texttt{ctx}(F) = \Gamma \\ \Gamma \vdash e : A \to A' \qquad \texttt{loopnest}(F) = \texttt{notinloop} \\ \texttt{notinloop} \vdash e\ \texttt{ok} \qquad \Sigma; \Gamma \vdash e\ \texttt{pending}\ \tau \to \mathbf{cmd} \ @\ \tau'\end{array}}{\Sigma; A \vdash F, e : \tau \to \tau'} \text{ CHECKFRAMETYPE-FRAMEEXPRET}$$

$$\frac{\Sigma; A - \{x\} \vdash F : \mathbf{cmd} \to \tau'}{\Sigma; A \vdash F, x : \tau = v : \mathbf{cmd} \to \tau'} \text{ CHECKFRAMETYPE-FRAMEVAL}$$

$$\frac{\Sigma; A - \{x\} \vdash F : \mathbf{cmd} \to \tau'}{\Sigma; A \vdash F, x : \tau_x : \mathbf{cmd} \to \tau'} \text{ CHECKFRAMETYPE-FRAMEVAR}$$

$$\frac{\begin{array}{c}\texttt{assigned}(F) = A' \qquad \Sigma; A' \vdash F : \mathbf{cmd} \to \tau' \\ \Sigma \vdash \texttt{ctx}(F) = \Gamma \qquad \Sigma; \Gamma \vdash \texttt{loop}(e_c, e) : \mathbf{cmd} \\ \Gamma \vdash \texttt{loop}(e_c, e) : A' \to A' \qquad \texttt{loopnest}(F) = L \\ L \vdash \texttt{loop}(e_c, e)\ \texttt{ok} \qquad \Sigma; \Gamma \vdash \texttt{loop}(e_c, e)\ \texttt{canreturn}\ \tau'\end{array}}{\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau'} \text{ CHECKFRAMETYPE-FRAMELOOP}$$

$\boxed{\Sigma \vdash \texttt{ctx}(F) = \Gamma}$

$$\frac{}{\Sigma \vdash \texttt{ctx}(\cdot) = \cdot} \text{ GETFRAMECONTEXT-EMPTY} \qquad \frac{\Sigma \vdash \texttt{ctx}(F) = \Gamma}{\Sigma \vdash \texttt{ctx}(F, e) = \Gamma} \text{ GETFRAMECONTEXT-EXP}$$

$$\frac{\Sigma \vdash \texttt{ctx}(F) = \Gamma}{\Sigma \vdash \texttt{ctx}(F, x : \tau) = \Gamma, x : \tau} \text{ GETFRAMECONTEXT-VAR}$$

$$\frac{\Sigma \vdash \texttt{ctx}(F) = \Gamma \qquad \Sigma \vdash v : \tau}{\Sigma \vdash \texttt{ctx}(F, x : \tau = v) = \Gamma, x : \tau} \text{ GETFRAMECONTEXT-VAL}$$

$$\frac{\Sigma \vdash \texttt{ctx}(F) = \Gamma}{\Sigma \vdash \texttt{ctx}(F, \texttt{loopctx}(e_c, e)) = \Gamma} \text{ GETFRAMECONTEXT-LOOP}$$

$\boxed{\texttt{assigned}(F) = A}$

$$\frac{}{\texttt{assigned}(\cdot) = \{\}}\ \textsc{getframeassigned-empty}$$

$$\frac{\texttt{assigned}(F) = A}{\texttt{assigned}(F, e) = A}\ \textsc{getframeassigned-exp}$$

$$\frac{\texttt{assigned}(F) = A}{\texttt{assigned}(F, x : \tau) = A}\ \textsc{getframeassigned-var}$$

$$\frac{\texttt{assigned}(F) = A \qquad x \notin A}{\texttt{assigned}(F, x : \tau = v) = A \cup \{x\}}\ \textsc{getframeassigned-val}$$

$$\frac{\texttt{assigned}(F) = A}{\texttt{assigned}(F, \texttt{loopctx}(e_c, e)) = A}\ \textsc{getframeassigned-loop}$$

$\boxed{\texttt{loopnest}(F) = L}$

$$\frac{}{\texttt{loopnest}(\cdot) = \texttt{ notinloop}}\ \textsc{getloopcontext-empty}$$

$$\frac{\texttt{loopnest}(F) = L}{\texttt{loopnest}(F, x : \tau) = L}\ \textsc{getloopcontext-decl}$$

$$\frac{\texttt{loopnest}(F) = L}{\texttt{loopnest}(F, x : \tau = v) = L}\ \textsc{getloopcontext-def}$$

$$\frac{\texttt{loopnest}(F) = L}{\texttt{loopnest}(F, e) = L}\ \textsc{getloopcontext-exp}$$

$$\frac{}{\texttt{loopnest}(F, \texttt{loopctx}(e_c, e)) = \texttt{ inloop}}\ \textsc{getloopcontext-loop}$$

$\boxed{\texttt{innerloop}(F) = F'}$

$$\frac{\texttt{innerloop}(F) = F'}{\texttt{innerloop}(F, x : \tau) = F'}\ \textsc{getinnerloop-decl}$$

$$\frac{\texttt{innerloop}(F) = F'}{\texttt{innerloop}(F, x : \tau = v) = F'}\ \textsc{getinnerloop-def}$$

$$\frac{\texttt{innerloop}(F) = F'}{\texttt{innerloop}(F, e) = F'}\ \textsc{getinnerloop-exp}$$

$$\frac{}{\texttt{innerloop}(F, \texttt{loopctx}(e_c, e)) = F, \texttt{loopctx}(e_c, e)}\ \text{GETINNERLOOP-LOOP}$$

$\boxed{\Sigma; \Gamma \vdash e\ \texttt{pending}\ \tau \to \tau'\ @\ \tau''}$

$$\frac{\begin{array}{c} op : \tau \times \tau'' \to \tau' \\ \Sigma; \Gamma \vdash e_2 : \tau'' \qquad \tau'' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_2\ \texttt{canreturn}\ \tau_r \end{array}}{\Sigma; \Gamma \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{pending}\ \tau \to \tau'\ @\ \tau_r}\ \text{ISPENDING-BINOPL}$$

$$\frac{op : \tau'' \times \tau \to \tau' \qquad \Sigma \vdash v' : \tau''}{\Sigma; \Gamma \vdash \texttt{binop}(op, v', \cdot)\ \texttt{pending}\ \tau \to \tau'\ @\ \tau''}\ \text{ISPENDING-BINOPR}$$

$$\frac{op : \tau \to \tau'}{\Sigma; \Gamma \vdash \texttt{monop}(op, \cdot)\ \texttt{pending}\ \tau \to \tau'\ @\ \tau''}\ \text{ISPENDING-MONOP}$$

$$\frac{\Sigma; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \texttt{call}(\cdot, e)\ \texttt{pending}\ (\tau \to \tau')* \to \tau'\ @\ \tau''}\ \text{ISPENDING-CALLF}$$

$$\frac{\Sigma; \Gamma \vdash \texttt{ptr}(a) : \tau \to \tau'*}{\Sigma; \Gamma \vdash \texttt{call}(\texttt{ptr}(a), \cdot)\ \texttt{pending}\ \tau \to \tau'\ @\ \tau''}\ \text{ISPENDING-CALLA}$$

$$\frac{\begin{array}{c} \Sigma \vdash v_1 : \tau_1 \qquad \ldots \qquad \Sigma \vdash v_{i-1} : \tau_{i-1} \qquad \Sigma; \Gamma \vdash e_{i+1} : \tau_{i+1} \\ \ldots \qquad \Sigma; \Gamma \vdash e_n : \tau_n \qquad \tau_1\ \texttt{small} \qquad \ldots \qquad \tau_n\ \texttt{small} \end{array}}{\Sigma; \Gamma \vdash (v_1, \ldots, v_{i-1}, \cdot, e_{i+1}, \ldots, e_n)\ \texttt{pending}\ \tau_i \to \texttt{tuple}(\tau_1, \ldots, \tau_n)\ @\ \tau''}\ \text{ISPENDING-TUPLE}$$

$$\frac{\begin{array}{c} \Sigma; \Gamma \vdash e_t : \tau' \qquad \Sigma; \Gamma \vdash e_f : \tau' \qquad \tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_t\ \texttt{canreturn}\ \tau'' \\ \tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_f\ \texttt{canreturn}\ \tau'' \end{array}}{\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{pending}\ \mathbf{bool} \to \tau'\ @\ \tau''}\ \text{ISPENDING-IF}$$

$$\frac{\Sigma; \Gamma \vdash x : \tau}{\Sigma; \Gamma \vdash \texttt{assign}(x, \cdot)\ \texttt{pending}\ \tau \to \mathbf{cmd}\ @\ \tau''}\ \text{ISPENDING-ASSIGN}$$

$$\frac{\tau\ \texttt{small}}{\Sigma; \Gamma \vdash \texttt{return}(\cdot)\ \texttt{pending}\ \tau \to \mathbf{cmd}\ @\ \tau}\ \text{ISPENDING-RETURN}$$

$$\frac{\tau_a\ \texttt{alloc}}{\Sigma; \Gamma \vdash \texttt{allocarray}(\tau_a, \cdot)\ \texttt{pending}\ \mathbf{int} \to \tau_a[]\ @\ \tau''}\ \text{ISPENDING-ALLOCARRAY}$$

$\boxed{\bowtie\ e\ \texttt{ok}}$

$$\frac{}{\triangleright\ e\ \texttt{ok}}\ \text{DIROK-PUSHING} \qquad \frac{}{\triangleleft\ v\ \texttt{ok}}\ \text{DIROK-RETURNING} \qquad \frac{}{\triangleleft\ (v_1, \ldots, v_n)\ \texttt{ok}}\ \text{DIROK-TUPLE}$$

$\boxed{\Sigma \vdash K : \tau \to \tau'}$

$$\frac{}{\Sigma \vdash \cdot : \tau \to \tau}\ \textsc{checkstack-empty}$$

$$\frac{\Sigma; A \vdash F : \tau \to \tau'' \qquad \texttt{assigned}(F) = A \qquad \Sigma \vdash K : \tau'' \to \tau'}{\Sigma \vdash K; F : \tau \to \tau'}\ \textsc{checkstack-nonempty}$$

$\boxed{\Sigma \vdash K \bowtie t : \tau}$

$$\frac{}{\Sigma \vdash K \lhd \texttt{exn} : \tau}\ \textsc{checkstate-exn} \qquad \frac{\Sigma; \cdot \vdash e : \tau \qquad \lhd\ e\ \texttt{ok}}{\Sigma \vdash \cdot \lhd e : \tau}\ \textsc{checkstate-empty}$$

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{ctx}(F) = \Gamma \qquad \Sigma; \Gamma \vdash e : \tau' \\ \texttt{assigned}(F) = A \qquad \Gamma \vdash e : A \to A' \qquad \texttt{loopnest}(F) = L \\ L \vdash e\ \texttt{ok} \qquad \Sigma; A' \vdash F : \tau' \to \tau'' \qquad \Sigma \vdash K : \tau'' \to \tau \\ \bowtie\ e\ \texttt{ok} \qquad \tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e\ \texttt{canreturn}\ \tau''\end{array}}{\Sigma \vdash K; F \bowtie e : \tau}\ \textsc{checkstate-normal}$$

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{ctx}(F) = \Gamma \\ \Sigma; \Gamma \vdash e : \mathbf{cmd} \qquad \texttt{assigned}(F) = A \qquad \Gamma \vdash e : A \to A' \\ \texttt{loopnest}(F) = \texttt{notinloop} \qquad \texttt{notinloop} \vdash e\ \texttt{ok} \\ \Sigma; \Gamma \vdash e\ \texttt{returns}\ \tau'' \qquad \Sigma \vdash K : \tau'' \to \tau \qquad \bowtie\ e\ \texttt{ok}\end{array}}{\Sigma \vdash K; F \bowtie e : \tau}\ \textsc{checkstate-returns}$$

$$\frac{\texttt{loopnest}(F) = \texttt{inloop} \qquad \texttt{innerloop}(F) = F' \qquad \texttt{assigned}(F') = A \qquad \Sigma; A \vdash F' : \mathbf{cmd} \to \tau'' \qquad \Sigma \vdash K : \tau'' \to \tau}{\Sigma \vdash K; F \lhd \texttt{break} : \tau}\ \textsc{checkstate-loopbrk}$$

$$\frac{\begin{array}{c}\texttt{loopnest}(F) = \texttt{inloop} \\ \texttt{innerloop}(F) = F' \qquad \texttt{assigned}(F') = A \\ \Sigma; A \vdash F' : \mathbf{cmd} \to \tau'' \qquad \Sigma \vdash K : \tau'' \to \tau\end{array}}{\Sigma \vdash K; F \lhd \texttt{continue} : \tau}\ \textsc{checkstate-loopcont}$$

$\boxed{\Sigma \vdash K \lhd t\ \texttt{final}}$

$$\frac{}{\Sigma \vdash \cdot \lhd \texttt{exn final}}\ \textsc{finalstate-exn} \qquad \frac{\Sigma; \cdot \vdash e : \tau \qquad \lhd\ e\ \texttt{ok}}{\Sigma \vdash \cdot \lhd e\ \texttt{final}}\ \textsc{finalstate-val}$$

$\boxed{\texttt{allocval}(\mu, a, \tau) = \mu'}$

Here we use the syntax $[\mu|a : v]$ to indicate a new function $\mu'(a') =$ if $a' = a$ then v else $\mu(a')$

$$\frac{}{\texttt{allocval}(\mu, a, \mathbf{bool}) = [\mu \mid a : \mathbf{false}]}\ \text{ALLOCVAL-BOOL}$$

$$\frac{}{\texttt{allocval}(\mu, a, \mathbf{int}) = [\mu \mid a : \overline{0}]}\ \text{ALLOCVAL-INT}$$

$$\frac{}{\texttt{allocval}(\mu, a, \tau*) = [\mu \mid a : \texttt{ptr}(\mathbf{null})]}\ \text{ALLOCVAL-PTR}$$

$$\frac{}{\texttt{allocval}(\mu, a, \tau[]) = [\mu \mid a : \texttt{array}(\mathbf{null}, 0)]}\ \text{ALLOCVAL-ARRAY}$$

$$\frac{}{\texttt{allocval}(\mu, a, \texttt{struct}(\cdot)) = [\mu \mid a : \texttt{rec}(a)]}\ \text{ALLOCVAL-EMPTYSTRUCT}$$

$$\frac{\texttt{allocval}(\mu, a, \texttt{struct}(p)) = \mu' \qquad \texttt{allocval}(\mu', a + x, \tau) = \mu''}{\texttt{allocval}(\mu, a, \texttt{struct}(p, f : \tau)) = \mu''}\ \text{ALLOCVAL-STRUCT}$$

$\boxed{\texttt{makearray}(\mu, a, \tau, n) = \mu'}$

$$\frac{\texttt{allocval}(\mu, a, \tau) = \mu'}{\texttt{makearray}(\mu, a, \tau, 0) = \mu'}\ \text{MAKEARRAY-EMPTY}$$

$$\frac{\texttt{makearray}(\mu, a, \tau, n) = \mu' \qquad \texttt{allocval}(\mu', a + n, \tau) = \mu''}{\texttt{makearray}(\mu, a, \tau, n + 1) = \mu''}\ \text{MAKEARRAY-NONEMPTY}$$

$\boxed{\mu \mid op(v_1, v_2) \to \mu' \mid t}$ Most of the rules for this judgment are elided for brevity as they have clear semantics from the prose description in Chapter 2.

$$\frac{}{\mu \mid \texttt{div}(\overline{n}, \overline{0}) \to \mu \mid \texttt{exn}} \qquad \frac{}{\mu \mid \texttt{mod}(\overline{n}, \overline{0}) \to \mu \mid \texttt{exn}} \qquad \frac{n > m}{\mu \mid \texttt{cmpg}(\overline{n}, \overline{m}) \to \mu \mid \mathbf{true}}$$

$$\frac{n \leq m}{\mu \mid \texttt{cmpg}(\overline{n}, \overline{m}) \to \mu \mid \mathbf{false}} \qquad \frac{}{\mu \mid \texttt{seq}(\texttt{nop}, \texttt{nop}) \to \mu \mid \texttt{nop}}$$

$$\frac{a \notin \texttt{dom}(\mu)}{\mu \mid \texttt{write}(\texttt{ptr}(a), v) \to \mu \mid \texttt{exn}} \qquad \frac{a \in \texttt{dom}(\mu)}{\mu \mid \texttt{write}(\texttt{ptr}(a), v) \to [\mu|\ a : v] \mid v}$$

$$\frac{0 < i < n}{\mu \mid \texttt{arrayindex}(\texttt{array}(a, n), \overline{i}) \to \mu \mid \texttt{ptr}(a + i)}$$

$$\frac{i < 0}{\mu \mid \texttt{arrayindex}(\texttt{array}(a, n), \overline{i}) \to \mu \mid \texttt{exn}} \qquad \frac{i \geq n}{\mu \mid \texttt{arrayindex}(\texttt{array}(a, n), \overline{i}) \to \mu \mid \texttt{exn}}$$

$\boxed{\mu \mid op(v) \to t}$ As with the previous judgment, many of the inference rules are omitted.

$$\frac{}{\mu \mid \texttt{lognot}(\textbf{false}) \to \textbf{true}} \qquad \frac{}{\mu \mid \texttt{lognot}(\textbf{true}) \to \textbf{false}} \qquad \frac{}{\mu \mid \texttt{ign}(v) \to \texttt{nop}}$$

$$\frac{a \notin \texttt{dom}(\mu)}{\mu \mid \texttt{read}(\texttt{ptr}(a)) \to \texttt{exn}} \qquad \frac{\mu(a) = v}{\mu \mid \texttt{read}(\texttt{ptr}(a)) \to v}$$

$$\frac{}{\mu \mid \texttt{field}(f)(\texttt{rec}(a)) \to \texttt{ptr}(a + f)}$$

$\boxed{\mu \mid K; F \bowtie t \to \mu' \mid K' \bowtie' t'}$

$$\frac{}{\mu \mid K; F \rhd v \to \mu \mid K; F \lhd v}\ \text{STEP-VAL}$$

$$\frac{}{\mu \mid K; F, x : \tau_x = v, \ldots \rhd x \to \mu \mid K; F, x : \tau_x = v, \ldots \lhd v}\ \text{STEP-VAR}$$

$$\frac{}{\mu \mid K; F \rhd \texttt{binop}(op, e_1, e_2) \to \mu \mid K; F, \texttt{binop}(op, \cdot, e_2) \rhd e_1}\ \text{STEP-PUSHBINOP}$$

$$\frac{}{\mu \mid K; F, \texttt{binop}(op, \cdot, e_2) \lhd v \to \mu \mid K; F, \texttt{binop}(op, v, \cdot) \rhd e_2}\ \text{STEP-SWAPBINOP}$$

$$\frac{\mu \mid op(v_1, v_2) \to \mu' \mid t}{\mu \mid K; F, \texttt{binop}(op, v_1, \cdot) \lhd v_2 \to \mu' \mid K; F \lhd t}\ \text{STEP-POPBINOP}$$

$$\frac{}{\mu \mid K; F \rhd \texttt{monop}(op, e) \to \mu \mid K; F, \texttt{monop}(op, \cdot) \rhd e}\ \text{STEP-PUSHMONOP}$$

$$\frac{\mu \mid op(v) \to t}{\mu \mid K; F, \texttt{monop}(op, \cdot) \lhd v \to \mu \mid K; F \lhd t}\ \text{STEP-POPMONOP}$$

$$\frac{}{\mu \mid K; F \rhd \texttt{call}(e_f, e) \to \mu \mid K; F, \texttt{call}(\cdot, e) \rhd e_f}\ \text{STEP-PUSHCALLFN}$$

$$\frac{}{\mu \mid K; F, \texttt{call}(\cdot, e) \lhd v \to \mu \mid K; F, \texttt{call}(v, \cdot) \rhd e}\ \text{STEP-PUSHCALLARGS}$$

$$\frac{\mu(a) = \texttt{func}(x_1, \ldots, x_n, e)}{\mu \mid K; F, \texttt{call}(\texttt{ptr}(a), \cdot) \lhd (v_1, \ldots, v_n) \to \mu \mid K; F; \cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n \rhd e}$$

STEP-FINALIZECALL

$$\frac{}{\mu \mid K; F, \texttt{call}(\texttt{ptr}(\textbf{null}), \cdot) \lhd (v_1, \ldots, v_n) \to \mu \mid K; F \lhd \texttt{exn}}\ \text{STEP-CALLNULL}$$

$$\frac{}{\mu \mid K; F \rhd () \to \mu \mid K; F \lhd ()} \text{ STEP-PUSHEMPTYTUPLE}$$

$$\frac{}{\mu \mid K; F \rhd (e_1, \dots) \to \mu \mid K; F, (\cdot, \dots) \rhd e_1} \text{ STEP-PUSHTUPLEELEM}$$

$$\frac{}{\mu \mid K; F, (\dots, \cdot, e_i, \dots) \lhd v \to \mu \mid K; F, (\dots, v, \cdot, \dots) \rhd e_i} \text{ STEP-NEXTTUPLEELEM}$$

$$\frac{}{\mu \mid K; F, (\dots, \cdot) \lhd v \to \mu \mid K; F \lhd (\dots, v)} \text{ STEP-LASTTUPLEELEM}$$

$$\frac{a \notin \mathtt{dom}(\mu) \qquad \mathtt{allocval}(\mu, a, \tau_a) = \mu'}{\mu \mid K; F \rhd \mathtt{alloc}(\tau_a) \to \mu' \mid K; F \lhd \mathtt{ptr}(a)} \text{ STEP-ALLOC}$$

$$\frac{}{\mu \mid K; F \rhd \mathtt{allocarray}(\tau_a, e) \to \mu' \mid K; F, \mathtt{allocarray}(\tau_a, \cdot) \rhd e} \text{ STEP-PUSHALLOCARRAY}$$

$$\frac{a \notin \mathtt{dom}(\mu) \qquad \mathtt{makearray}(\mu, a, \tau_a, n) = \mu' \qquad n \geq 0}{\mu \mid K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} \to \mu' \mid K; F \lhd \mathtt{array}(a, n)} \text{ STEP-POPALLOCARRAY}$$

$$\frac{n < 0}{\mu \mid K; F, \mathtt{allocarray}(\tau, \cdot) \lhd \overline{n} \to \mu \mid K; F \lhd \mathtt{exn}} \text{ STEP-POPALLOCARRAYERR}$$

$$\frac{}{\mu \mid K; F \rhd \mathtt{if}(e_c, e_t, e_f) \to \mu \mid K; F, \mathtt{if}(\cdot, e_t, e_f) \rhd e_c} \text{ STEP-PUSHIF}$$

$$\frac{}{\mu \mid K; F, \mathtt{if}(\cdot, e_t, e_f) \lhd \mathbf{true} \to \mu \mid K; F \rhd e_t} \text{ STEP-POPIFTRUE}$$

$$\frac{}{\mu \mid K; F, \mathtt{if}(\cdot, e_t, e_f) \lhd \mathbf{false} \to \mu \mid K; F \rhd e_f} \text{ STEP-POPIFFALSE}$$

$$\frac{}{\mu \mid K; F \rhd \mathtt{decl}(x, \tau, e) \to \mu \mid K; F, x : \tau \rhd e} \text{ STEP-PUSHDECL}$$

$$\frac{}{\mu \mid K; F, x : \tau \lhd \mathtt{nop} \to \mu \mid K; F \lhd \mathtt{nop}} \text{ STEP-POPDECL}$$

$$\frac{}{\mu \mid K; F \rhd \mathtt{assign}(x, e) \to \mu \mid K; F, \mathtt{assign}(x, \cdot) \rhd e} \text{ STEP-PUSHASSIGN}$$

$$\frac{}{\mu \mid K; F, x : \tau_x = v, \dots, \mathtt{assign}(x, \cdot) \lhd v' \to \mu \mid K; F, x : \tau_x = v', \dots \lhd \mathtt{nop}} \text{ STEP-POPASSIGN}$$

$$\frac{}{\mu \mid K; F, x : \tau_x, \dots, \mathtt{assign}(x, \cdot) \lhd v' \to \mu \mid K; F, x : \tau_x = v', \dots \lhd \mathtt{nop}} \text{ STEP-POPASSIGNFIRST}$$

$$\frac{}{\mu \mid K; F, x : \tau_x = v \lhd \mathtt{nop} \to \mu \mid K; F \lhd \mathtt{nop}} \text{ STEP-POPASSIGNED}$$

$$\frac{}{\mu \mid K; F \rhd \texttt{return}(e) \to \mu \mid K; F, \texttt{return}(\cdot) \rhd e} \text{ STEP-PUSHRET}$$

$$\frac{}{\mu \mid K; F, \texttt{return}(\cdot) \lhd e \to \mu \mid K \lhd e} \text{ STEP-POPRET}$$

$$\frac{}{\mu \mid K; F \rhd \texttt{loop}(e_c, e) \to \mu \mid K; F, \texttt{loopctx}(e_c, e) \rhd \texttt{if}(e_c, e, \texttt{break})} \text{ STEP-LOOP}$$

$$\frac{}{\mu \mid K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{nop} \to \mu \mid K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{continue}} \text{ STEP-LOOPPOP}$$

$$\frac{}{\mu \mid K; F \rhd \texttt{break} \to \mu \mid K; F \lhd \texttt{break}} \text{ STEP-BREAK}$$

$$\frac{}{\mu \mid K; F, x : \tau_x = v \lhd \texttt{break} \to \mu \mid K; F \lhd \texttt{break}} \text{ STEP-BREAKVAL}$$

$$\frac{}{\mu \mid K; F, x : \tau \lhd \texttt{break} \to \mu \mid K; F \lhd \texttt{break}} \text{ STEP-BREAKVAR}$$

$$\frac{}{\mu \mid K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{break} \to \mu \mid K; F \lhd \texttt{nop}} \text{ STEP-BREAKLOOP}$$

$$\frac{}{\mu \mid K; F, e \lhd \texttt{break} \to \mu \mid K; F \lhd \texttt{break}} \text{ STEP-BREAKEXP}$$

$$\frac{}{\mu \mid K; F \rhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}} \text{ STEP-CONTINUE}$$

$$\frac{}{\mu \mid K; F, x : \tau_x = v \lhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}} \text{ STEP-CONTINUEVAL}$$

$$\frac{}{\mu \mid K; F, x : \tau \lhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}} \text{ STEP-CONTINUEVAR}$$

$$\frac{}{\mu \mid K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{continue} \to \mu \mid K; F \rhd \texttt{loop}(e_c, e)} \text{ STEP-CONTINUELOOP}$$

$$\frac{}{\mu \mid K; F, e \lhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}} \text{ STEP-CONTINUEEXP}$$

$$\frac{}{\mu \mid K; F \lhd \texttt{exn} \to \mu \mid \cdot \lhd \texttt{exn}} \text{ STEP-EXNPROP}$$

$\boxed{\mu : \Sigma}$

$$\frac{\texttt{dom}(\mu) = \texttt{dom}(\Sigma) \qquad \forall a \in \texttt{dom}(\mu).\mu(a) = v \wedge \Sigma \vdash v : \tau \wedge \Sigma(a) = \tau}{\mu : \Sigma}$$

$\boxed{\Sigma \leq \Sigma'}$

$$\frac{\forall a \in \texttt{dom}(\Sigma).\Sigma(a) = \Sigma'(a)}{\Sigma \leq \Sigma'}$$

A.3 Safety

A.3.1 Progress

If $\mu : \Sigma$ and $\Sigma \vdash K \bowtie t : \tau$ then either $\Sigma \vdash K \lhd t$ `final` or $\mu \mid K \bowtie t \to \mu' \mid K' \bowtie' t'$ for some μ', K', $\bowtie'$, and t'.

A.3.2 Preservation

If $\mu : \Sigma$ and $\Sigma \vdash K \bowtie t : \tau$ and $\mu \mid K \bowtie t \to \mu' \mid K' \bowtie' t'$ then $\exists \Sigma'.\mu' : \Sigma'$ and $\Sigma' \vdash K' \bowtie t' : \tau$ and $\Sigma \leq \Sigma'$.

A.4 Lemmas

1. If $\texttt{assigned}(F) = A$ and $x \in A$ then F has the form $F', x : \tau_x = v, \ldots$.
   Proof is by induction on the derivation of $\texttt{assigned}(F) = A$.
2. If $\Sigma \vdash v : \tau$ then
   (a) If $\tau = \mathbf{cmd}$ then $v = \texttt{nop}$.
   (b) If $\tau = \mathbf{bool}$ then either $v = \mathbf{true}$ or $v = \mathbf{false}$.
   (c) If $\tau = \mathbf{int}$ then $v = \overline{n}$ for some $n \in [-2^{31}, 2^{31})$.
   (d) If $\tau = \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau$ then $v = \texttt{func}(x_1, \ldots, x_n, e)$ where $\Gamma = \cdot, x_1 : \tau_1, \ldots, x_n : \tau_n$, $A = \{x_1, \ldots, x_n\}$, $\Sigma; \Gamma \vdash e : \mathbf{cmd}$, $\Gamma \vdash e : A \to A$, $\texttt{notinloop} \vdash e$ `ok` , $\Sigma; \Gamma \vdash e$ `returns` τ, τ `small` , and τ_1 `small` ... τ_n `small` .
   (e) If $\tau = \tau*$ then either $v = \texttt{ptr}(\mathbf{null})$ or $v = \texttt{ptr}(a)$ where $\Sigma(a) = \tau$.
   (f) If $\tau = \tau[]$ then $v = \texttt{array}(a, n)$ such that $\forall i \in [0, n).\Sigma(a + i) = \tau$.
   (g) If $\tau = \texttt{struct}(p)$ then $v = \texttt{rec}(a)$ where $\forall f : \tau \in p.\Sigma(a + f) = \tau$.
   Proof is by inversion on the rules for $\Sigma \vdash v : \tau$.
3. If $\mu : \Sigma$ and $op : \tau_1 \times \tau_2 \to \tau$ and $\Sigma \vdash v_1 : \tau_1$ and $\Sigma \vdash v_2 : \tau_2$ then $\mu \mid op(v_1, v_2) \to \mu' \mid t$.
   Proof by case analysis on the derivation of $op : \tau_1 \times \tau_2 \to \tau$.
4. If $\mu : \Sigma$ and $op : \tau \to \tau'$ and $\Sigma \vdash v : \tau$ then $\mu \mid op(v) \to t$.
   Proof by case analysis of the derivation of $op : \tau \to \tau'$.
5. If $\Sigma \vdash \texttt{ctx}(F) = \Gamma, x : \tau$ then either F has the form $F', x : \tau, \ldots$ or the form $F', x : \tau = v, \ldots$ where $\Sigma \vdash v : \tau$.
   Proof by case analysis of the derivation of $\Sigma \vdash \texttt{ctx}(F) = \Gamma, x : \tau$.
6. If τ `alloc` and $a \notin \texttt{dom}(\mu)$ then $\exists \mu'.\texttt{allocval}(\mu, a, \tau) = \mu'$.
   Proof by induction on τ.
7. If τ `alloc` and $a \notin \texttt{dom}(\mu)$ and $n \geq 0$ then $\exists \mu'.\texttt{makearray}(\mu, a, \tau, n) = \mu'$.
   Proof by induction on n.

8. If τ `alloc` and $a \notin \mathtt{dom}(\mu)$ and $\mathtt{allocval}(\mu, a, \tau) = \mu'$ then $\exists \Sigma'.\mu' : \Sigma'$ and $\Sigma \leq \Sigma'$ and $\Sigma'(a) = \tau$.

   Proof by induction on τ.

9. If τ `alloc` and $a \notin \mathtt{dom}(\mu)$ and $n \geq 0$ and $\mathtt{makearray}(\mu, a, \tau, n) = \mu'$ then $\exists \Sigma'.\mu' : \Sigma'$ and $\Sigma \leq \Sigma'$ and $\Sigma' \vdash \mathtt{array}(a, n) : \tau[]$.

   Proof by induction on n.

10. If $\Sigma \vdash K \bowtie e : \tau$ and $\Sigma \leq \Sigma'$ then $\Sigma' \vdash K \bowtie e : \tau$.

    Proof is by induction on the derivation of $\Sigma \vdash K \bowtie e : \tau$ with lots of auxiliary induction.

11. If $\Sigma; \Gamma \vdash e$ `pending` $\mathbf{cmd} \to \tau \mathbin{@} \tau''$ then $\tau = \mathbf{cmd}$ and $e = \mathtt{binop}(\mathtt{seq}, e_1, e_2)$.

    Proof by case analysis on the derivation of $\Sigma; \Gamma \vdash e$ `pending` $\mathbf{cmd} \to \tau \mathbin{@} \tau''$

    $$\frac{\begin{array}{ll} & op : \mathbf{cmd} \times \tau'' \to \tau \\ \Sigma; \Gamma \vdash e_2 : \tau'' & \tau'' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_2 \ \mathtt{canreturn}\ \tau_r \end{array}}{\Sigma; \Gamma \vdash \mathtt{binop}(op, \cdot, e_2)\ \mathtt{pending}\ \mathbf{cmd} \to \tau \mathbin{@} \tau_r}$$

    $op = \mathtt{seq}$ and $\tau = \mathbf{cmd}$ — Inversion on $op : \mathbf{cmd} \times \tau'' \to \tau$

    $$\frac{op : \tau'' \times \mathbf{cmd} \to \tau \qquad \Sigma \vdash v' : \tau''}{\Sigma; \Gamma \vdash \mathtt{binop}(op, v', \cdot)\ \mathtt{pending}\ \mathbf{cmd} \to \tau \mathbin{@} \tau''}$$

    $op = \mathtt{seq}$ and $\tau = \mathbf{cmd}$ — Inversion on $op : \mathbf{cmd} \times \tau'' \to \tau$

    No other cases can occur.

12. If $\mathtt{loopnest}(F) =$ `inloop` and $\Sigma; A \vdash F : \mathbf{cmd} \to \tau$ then $\mathtt{innerloop}(F) = F', \mathtt{loopctx}(e_c, e)$, $\mathtt{assigned}(F') = A'$ and $\Sigma; A' \vdash F' : \mathbf{cmd} \to \tau$ for some e_c, e, F', and A'.

    Proof by induction on the structure of F:

    $F = F_0, x : \tau_x$
    - $\Sigma; A' - \{x\} \vdash F_0 : \mathbf{cmd} \to \tau$ — Inversion on $\Sigma; A \vdash F : \mathbf{cmd} \to \tau$
    - $\mathtt{loopnest}(F_0) = F', \mathtt{loopctx}(e_c, e)$ and $\mathtt{assigned}(F') = A'$ and $\Sigma; A' \vdash F' : \mathbf{cmd} \to \tau$ — By inductive hypothesis
    - $\mathtt{loopnest}(F) = F', \mathtt{loopctx}(e_c, e)$ — Rule GETLOOPCONTEXT-DECL

    $F = F_0, x : \tau_x = v$
    - $\Sigma; A' - \{x\} \vdash F_0 : \mathbf{cmd} \to \tau$ — Inversion on $\Sigma; A \vdash F : \mathbf{cmd} \to \tau$
    - $\mathtt{innerloop}(F_0) = F', \mathtt{loopctx}(e_c, e)$ and $\mathtt{assigned}(F') = A'$ and $\Sigma; A' \vdash F' : \mathbf{cmd} \to \tau$ — By inductive hypothesis
    - $\mathtt{loopnest}(F) = F', \mathtt{loopctx}(e_c, e)$ — Rule GETLOOPCONTEXT-DEF

    $F = F_0, e$

$\Gamma \vdash e : A \to A''$ and $\Sigma; \Gamma \vdash e$ `pending` $\mathbf{cmd} \to \tau' \mathbin{@} \tau''$

By inversion on $\Sigma; A \vdash F : \mathbf{cmd} \to \tau$

$\tau' = \mathbf{cmd}$ By Lemma 11

$\Sigma; A'' \vdash F : \mathbf{cmd} \to \tau$ By inversion on $\Sigma; A \vdash F : \mathbf{cmd} \to \tau$

$\texttt{loopnest}(F_0) = \texttt{inloop}$

By inversion on the derivation of $\texttt{loopnest}(F) = \texttt{inloop}$

$\texttt{innerloop}(F_0) = F', \texttt{loopctx}(e_c, e)$ and $\texttt{assigned}(F') = A'$

and $\Sigma; A' \vdash F' : \mathbf{cmd} \to \tau$ By inductive hypothesis

$\texttt{innerloop}(F) = F', \texttt{loopctx}(e_c, e)$ Rule GETINNERLOOP-EXP

$F = F_0, \texttt{loopctx}(e_c, e)$

$\texttt{innerloop}(F) = F_0, \texttt{loopctx}(e_c, e)$ Rule GETINNERLOOP-LOOP

$\texttt{assigned}(F') = A'$ and $\Sigma; A' \vdash F : \mathbf{cmd} \to \tau$ Inversion on $\Sigma; A \vdash F : \mathbf{cmd} \to \tau$

13. If $op : \tau' \times \tau'' \to \tau$ and $\mu : \Sigma$ and $\mu \mid op(v_1, v_2) \to \mu' \mid t$ then $\mu' : \Sigma$ and either $t = \texttt{exn}$ or else $t = v$ and $\Sigma \vdash v : \tau$.

    Proof by case analysis on the derivation of $\mu \mid op(v_1, v_2) \to \mu' \mid t$

14. If $op : \tau' \to \tau$ and $\mu : \Sigma$ and $\mu \mid op(v) \to t$ then either $t = \texttt{exn}$ or else $t = v$ and $\Sigma \vdash v : \tau$.

    Proof by case analysis on the derivation of $\mu \mid op(v) \to t$

15. If $\Gamma = \cdot, x_1 : \tau_1, \ldots, x_n : \tau_n$ and $\forall i \in [1, n]. \Sigma \vdash v_i : \tau_i$ and $F = \cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n$ then $\Sigma \vdash \texttt{ctx}(F) = \Gamma$.

    Proof by induction on the structure of F.

16. If $A = \{x_1, \ldots, x_n\}$ and $F = \cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n$ then $\texttt{assigned}(F) = A$.

    Proof by induction on the structure of F.

17. If $\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : A_1 \to A_2$ and $A_1 \subseteq A_1' \subseteq \{x_1, \ldots, x_n\}$ then $\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : A_1' \to A_2'$ where $A_2 \subseteq A_2' \subseteq \{x_1, \ldots, x_n\}$.

    Proof by induction on the derivation of $\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : A_1 \to A_2$.

18. If $\Sigma; A \vdash F : \tau \to \tau'$ and $\Sigma \vdash \texttt{ctx}(F) = \cdot, x_1 : \tau_1, \ldots, x_n : \tau_n$ and $A \subseteq A' \subseteq \{x_1, \ldots, x_n\}$ then $\Sigma; A' \vdash F : \tau \to \tau'$.

    Proof by induction on the derivation of $\Sigma; A \vdash F : \tau \to \tau'$.

19. If $F = F_0, x : \tau_x, \ldots$ and $\Sigma \vdash v : \tau_x$ and $F' = F_0, x : \tau_x = v, \ldots$ and $\texttt{assigned}(F) = A$ then $\texttt{assigned}(F') = A \cup \{x\}$.

    Proof by induction on the structure of F.

20. If $F = F_0, x = v', \ldots$ and $\Sigma \vdash v' : \tau_v$ and $\Sigma \vdash v : \tau_v$ and $F' = F_0, x : \tau_x = v, \ldots$ and $\texttt{assigned}(F) = A$ then $\texttt{assigned}(F') = A$.

    Proof by induction on the structure of F.

21. If $F = F_0, x : \tau_x, \ldots$ and $\Sigma \vdash v : \tau_x$ and $F' = F_0, x : \tau_x = v, \ldots$ and $\Sigma \vdash \texttt{ctx}(F) = \Gamma$

then $\Sigma \vdash \texttt{ctx}(F') = \Gamma$.

Proof by induction on the structure of F.

22. If $F = F_0, x : \tau_x = v', \ldots$ and $\Sigma \vdash v' : \tau_v$ and $\Sigma \vdash v : \tau_v$ and $F' = F_0, x : \tau_x = v, \ldots$ and $\Sigma \vdash \texttt{ctx}(F) = \Gamma$ then $\Sigma \vdash \texttt{ctx}(F') = \Gamma$.

    Proof by induction on the structure of F.

23. If $F = F_0, x : \tau_x, \ldots$ and $\Sigma \vdash v : \tau_x$ and $F' = F_0, x : \tau_x = v, \ldots$ and $\Sigma; A \vdash F : \tau \to \tau'$ then $\Sigma; A \vdash F' : \tau \to \tau'$.

    Proof by induction on the structure of F.

24. If $F = F_0, x : \tau_x = v', \ldots$ and $\Sigma \vdash v' : \tau_x$ and $\Sigma \vdash v : \tau_x$ and $F' = F_0, x : \tau_x = v, \ldots$ and $\Sigma; A \vdash F : \tau \to \tau'$ then $\Sigma; A \vdash F' : \tau \to \tau'$.

    Proof by induction on the structure of F.

25. If $\Gamma \vdash e : A \to A'$ then $A \subseteq A'$.

    Proof by induction on the derivation of $\Gamma \vdash e : A \to A'$.

26. If $L \vdash e\ \texttt{ok}$ then $\texttt{inloop} \vdash e\ \texttt{ok}$.

    Proof by induction on the derivation of $L \vdash e\ \texttt{ok}$.

A.5 Proofs

A.5.1 Progress

Proof by case analysis on the derivation of $\Sigma \vdash K \bowtie t : \tau$.

1. Case CHECKSTATE-EXN was used so $t = \texttt{exn}$

   Via case analysis of the structure of K either

   (a) Case $K = \cdot$

   $\Sigma \vdash \cdot \lhd \texttt{exn}\ \texttt{final}$ Rule FINALSTATE-EXN

   (b) Case $K = K'; F$

   $\mu \mid K'; F \lhd \texttt{exn} \to \mu \mid \cdot \lhd \texttt{exn}$ Rule STEP-EXNPROP

2. Case CHECKSTATE-EMPTY was used so $K = \cdot$

   $\Sigma \vdash \cdot \lhd e\ \texttt{final}$ Rule FINALSTATE-VAL

3. Case CHECKSTATE-NORMAL was used so $K = K'; F$ and $t = e$ and $\Sigma; \Gamma \vdash e : \tau'$
   We now do a case analysis on $\Sigma; \Gamma \vdash e : \tau'$:

   (a) Case CHECKEXP-VAR was used so $e = x$

   $x \in A$ Inversion on $\Gamma \vdash x : A \to A$

$F = F', x : \tau_x = v, \ldots$ Lemma 1
$\mu \mid K'; F', x : \tau_x = v, \ldots \rhd x \to \mu \mid K'; F', x : \tau_x = v, \ldots \lhd v$ Rule STEP-VAR

(b) Case CHECKEXP-VALUE was used so $e = v$

Via case analysis of $\bowtie\ e\ \texttt{ok}$ we know that either DIROK-PUSHING or DIROK-RETURNING was used.

In the former case, STEP-VAL applies. For the latter, we must case analyze $\Sigma; A' \vdash F : \tau' \to \tau''$:

i. Case CHECKFRAMETYPE-FRAMELOOP was used so $F = F', \texttt{loopctx}(e_c, e')$ for some F', e_c and e'

$v = \texttt{nop}$ Lemma 2
$\mu \mid K'; F', \texttt{loopctx}(e_c, e') \lhd \texttt{nop} \to \mu \mid K'; F', \texttt{loopctx}(e_c, e') \lhd \texttt{continue}$
Rule STEP-LOOPPOP

ii. Case CHECKFRAMETYPE-FRAMEVAR was used so $F = F', x : \tau_x$ for some F', x and τ_x

$v = \texttt{nop}$ Lemma 2
$\mu \mid K'; F', x : \tau \lhd \texttt{nop} \to \mu \mid K'; F' \lhd \texttt{nop}$ Rule STEP-POPDECL

iii. Case CHECKFRAMETYPE-FRAMEVAL was used so $F = F', x : \tau_x = v$ for some F', x, τ_x and v

$v = \texttt{nop}$ Lemma 2
$\mu \mid K'; F', x : \tau_x = v \lhd \texttt{nop} \to \mu \mid K'; F' \lhd \texttt{nop}$ Rule STEP-POPASSIGNED

iv. Case CHECKFRAMETYPE-FRAMEEXPRET was used so $F = F', e'$ for some F' and e'
We must now case analysis on the derivation of $\vdash e'\ \texttt{returns}$:

A. Case DOESRETURN-RETURN was used so $e' = \texttt{return}(e'')$

$e'' = \cdot$ Inversion on $\Sigma; \Gamma \vdash e'\ \texttt{pending}\ \tau' \to \mathbf{cmd} \mathbin{@} \tau''$
$\mu \mid K'; F', \texttt{return}(\cdot) \lhd e' \to \mu \mid K' \lhd e'$ Rule STEP-POPRET

B. Case DOESRETURN-IF was used so $e' = \texttt{if}(e_c, e_t, e_f)$

$e_c = \cdot$ and $\tau' = \mathbf{bool}$ Inversion on $\Sigma; \Gamma \vdash e'$ `pending` $\tau' \to \mathbf{cmd} \mathbin{@} \tau''$
$v = \mathbf{true}$ or $v = \mathbf{false}$ Lemma 2
If $v = \mathbf{true}$ then
$\mu \mid K'; F', \mathtt{if}(\cdot, e_t, e_f) \lhd \mathbf{true} \to \mu \mid K'; F' \rhd e_t$
Rule STEP-POPIFTRUE
Otherwise $v = \mathbf{false}$ so
$\mu \mid K'; F', \mathtt{if}(\cdot, e_t, e_f) \lhd \mathbf{false} \to \mu \mid K'; F' \rhd e_f$
Rule STEP-POPIFFALSE

C. Case DOESRETURN-BINOPRHS was used so $e' = \mathtt{binop}(\mathtt{seq}, e_1, e_2)$

$e_2 \neq \cdot$ Via case analysis we know there is no derivation of $\vdash \cdot$ `returns`
$e_1 = \cdot$ Inversion on $\Sigma; \Gamma \vdash e'$ `pending` $\tau' \to \mathbf{cmd} \mathbin{@} \tau''$
$\mu \mid K'; F', \mathtt{binop}(op, \cdot, e_2) \lhd v \to \mu \mid K'; F', \mathtt{binop}(op, v, \cdot) \rhd e_2$
Rule STEP-SWAPBINOP

D. Case DOESRETURN-BINOPLHS was used so $e' = \mathtt{binop}(\mathtt{seq}, e_1, e_2)$

$e_1 \neq \cdot$ Via case analysis we know there is no derivation of $\vdash \cdot$ `returns`
$e_2 = \cdot$ and $e_1 = v'$ for some v'
Inversion on $\Sigma; \Gamma \vdash e'$ `pending` $\tau' \to \mathbf{cmd} \mathbin{@} \tau''$
This case does not occur.
Via case analysis we know there is no derivation of $\vdash v'$ `returns`

E. Case DOESRETURN-DECL was used so $e' = \mathtt{decl}(x, \tau_x, e'')$
This case does not occur because via case analysis we know there is no derivation of $\Sigma; \Gamma \vdash e'$ `pending` $\tau' \to \mathbf{cmd} \mathbin{@} \tau''$

v. Case CHECKFRAMETYPE-FRAMEEXPNORET was used so $F = F', e'$ for some F' and e'
We now do a case analysis of $\Sigma; \Gamma \vdash e'$ `pending` $\tau' \to \tau''' \mathbin{@} \tau''$

A. Case ISPENDING-BINOPL was used so $e' = \mathtt{binop}(op, \cdot, e_2)$

$\mu \mid K'; F', \mathtt{binop}(op, \cdot, e_2) \lhd v \to \mu \mid K'; F', \mathtt{binop}(op, v, \cdot) \rhd e_2$
Rule STEP-SWAPBINOP

B. Case ISPENDING-BINOPR was used so $e' = \mathtt{binop}(op, v', \cdot)$

$\mu \mid op(v, v') \to \mu' \mid t$ Lemma 3
$\mu \mid K'; F', \mathtt{binop}(op, v, \cdot) \lhd v' \to \mu' \mid K'; F' \lhd t$ Rule STEP-POPBINOP

C. Case ISPENDING-MONOP was used so $e' = \texttt{monop}(op, \cdot)$

$\mu \mid op(v) \to t$ Lemma 4
$\mu \mid K'; F', \texttt{monop}(op, \cdot) \lhd v \to \mu \mid K'; F' \lhd t$ Rule STEP-POPMONOP

D. Case ISPENDING-CALLF was used so $e' = \texttt{call}(\cdot, e_a)$

$\mu \mid K'; F', \texttt{call}(\cdot, e') \lhd v \to \mu \mid K'; F', \texttt{call}(v, \cdot) \rhd e'$
Rule STEP-PUSHCALLARGS

E. Case ISPENDING-TUPLE was used so $e' = (v_1, \ldots, v_{i-1}, \cdot, e_{i+1}, \ldots, e_n)$
Case analysis on the structure of $(v_1, \ldots, v_{i-1}, \cdot, e_{i+1}, \ldots, e_n)$

- $n = i$
  $\mu \mid K'; F', (\ldots, \cdot) \lhd v \to \mu \mid K'; F' \lhd (\ldots, v)$
  Rule STEP-LASTTUPLEELEM

- $n > i$
  $\mu \mid K'; F', (\ldots, \cdot, e_i, \ldots) \lhd v \to \mu \mid K'; F', (\ldots, v, \cdot, \ldots) \rhd e_i$
  Rule STEP-NEXTTUPLEELEM

F. Case ISPENDING-IF was used so $e' = \texttt{if}(\cdot, e_t, e_f)$
Clearly $\tau' = \mathbf{bool}$.

$v = \mathbf{true}$ or $v = \mathbf{false}$ Lemma 2
If $v = \mathbf{true}$ then
$\mu \mid K'; F', \texttt{if}(\cdot, e_t, e_f) \lhd \mathbf{true} \to \mu \mid K'; F' \rhd e_t$
Rule STEP-POPIFTRUE
Otherwise $v = \mathbf{false}$ so
$\mu \mid K'; F', \texttt{if}(\cdot, e_t, e_f) \lhd \mathbf{false} \to \mu \mid K'; F' \rhd e_f$
Rule STEP-POPIFFALSE

G. Case ISPENDING-ASSIGN was used so $e' = \texttt{assign}(x, \cdot)$

We know that either F' has the form $F'', x : \tau', \ldots$ or it has the form $F'', x : \tau_x = v', \ldots$ Lemma 5
In the former case,
$\mu \mid K'; F'', x : \tau', \ldots, \texttt{assign}(x, \cdot) \lhd v \to$
$\mu \mid K'; F'', x : \tau_x = v, \ldots \lhd \texttt{nop}$ Rule STEP-POPASSIGNFIRST
In the latter case,
$\mu \mid K'; F'', x : \tau' = v', \ldots, \texttt{assign}(x, \cdot) \lhd v \to$

$\mu \mid K'; F'', x : \tau_x = v, \ldots \lhd \texttt{nop}$ Rule STEP-POPASSIGN

H. Case ISPENDING-RETURN was used so $e' = \texttt{return}(e'')$

$\mu \mid K'; F', \texttt{return}(\cdot) \lhd e' \to \mu \mid K' \lhd e'$ Rule STEP-POPRET

I. Case ISPENDING-ALLOCARRAY was used so $e' = \texttt{allocarray}(\tau_a, \cdot)$

$v = \overline{n}$ Lemma 2
If $n \geq 0$ then
$\texttt{makearray}(\mu, a, \tau_a, n) = \mu'$ Lemma 7
$\mu \mid K'; F', \texttt{allocarray}(\tau_a, \cdot) \lhd \overline{n} \to \mu' \mid K'; F' \lhd \texttt{array}(a, n)$
Rule STEP-POPALLOCARRAY
Otherwise, $\mu \mid K'; F', \texttt{allocarray}(\tau, \cdot) \lhd \overline{n} \to \mu \mid K'; F' \lhd \texttt{exn}$
Rule STEP-POPALLOCARRAYERR

(c) Case CHECKEXP-BINOP was used so $e = \texttt{binop}(op, e_1, e_2)$

$\bowtie = \rhd$ Inversion on $\bowtie \texttt{binop}(op, e_1, e_2)\ \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{binop}(op, e_1, e_2) \to \mu \mid K'; F, \texttt{binop}(op, \cdot, e_2) \rhd e_1$
Rule STEP-PUSHBINOP

(d) Case CHECKEXP-MONOP was used so $e = \texttt{monop}(op, e')$

$\bowtie = \rhd$ Inversion on $\bowtie \texttt{monop}(op, e_a)\ \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{monop}(op, e) \to \mu \mid K'; F, \texttt{monop}(op, \cdot) \rhd e$ Rule STEP-PUSHMONOP

(e) Case CHECKEXP-TUPLE was used so $e = (e_1, \ldots, e_n)$

We do a case analysis on $\bowtie (e_1, \ldots, e_n)\ \texttt{ok}$:

i. Case DIROK-PUSHING was used so $\bowtie = \rhd$

If $n = 0$ then
$\mu \mid K'; F \rhd () \to \mu \mid K'; F \lhd ()$ Rule STEP-PUSHEMPTYTUPLE
Otherwise $\mu \mid K'; F \rhd (e_1, \ldots) \to \mu \mid K'; F, (\cdot, \ldots) \rhd e_1$
Rule STEP-PUSHTUPLEELEM

ii. Case DIROK-RETURNING was used so $\bowtie = \lhd$ and $e = v$

Does not occur since there is no derivation of $\Sigma \vdash v : \texttt{tuple}(\tau_1, \ldots, \tau_n)$.

iii. Case DIROK-TUPLE was used so $\bowtie = \lhd$ and $e = (v_1, \ldots, v_n)$
We do a case analysis on the derivation of $\Sigma; A' \vdash F' : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau''$

A. Case CHECKFRAMETYPE-FRAMEEXPNORET was used so $F' = F'', e'$

There are three possible derivations of
$\Sigma; \Gamma \vdash e' \texttt{ pending } (\tau_1, \ldots, \tau_n) \to \tau''' \mathbin{@} \tau''$.
Case ISPENDING-RETURN:
$e' = \texttt{return}(\cdot)$ and $\texttt{tuple}(\tau_1, \ldots, \tau_n)$ `small`
Inversion on $\Sigma; \Gamma \vdash e' \texttt{ pending } (\tau_1, \ldots, \tau_n) \to \tau''' \mathbin{@} \tau''$.
$n = 0$ Inversion on $\texttt{tuple}(\tau_1, \ldots, \tau_n)$ `small`
$\mu \mid K'; F', \texttt{return}(\cdot) \lhd () \to \mu \mid K' \lhd ()$ Rule STEP-POPRET
Case ISPENDING-MONOP where $op = \texttt{ign}$:
$e' = \texttt{monop}(\texttt{ign}, \cdot,)$ and $\texttt{ign} : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \mathbf{cmd}$
Inversion on $\Sigma; \Gamma \vdash e' \texttt{ pending } (\tau_1, \ldots, \tau_n) \to \tau''' \mathbin{@} \tau''$.
$\texttt{tuple}(\tau_1, \ldots, \tau_n)$ `small`
Inversion on $\texttt{ign} : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \mathbf{cmd}$
$n = 0$ Inversion on $\texttt{tuple}(\tau_1, \ldots, \tau_n)$ `small`
$\mu \mid \texttt{ign}(()) \to t$ Lemma 4
$\mu \mid K'; F', \texttt{monop}(op, \cdot) \lhd v \to \mu \mid K'; F' \lhd t$
Rule STEP-POPMONOP
Otherwise ISPENDING-CALLA was used:
$e' = \texttt{call}(\texttt{ptr}(a), \cdot)$ and $\Sigma \vdash \texttt{ptr}(a) : (\texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau''')*$
Inversion on $\Sigma; \Gamma \vdash e' \texttt{ pending } (\tau_1, \ldots, \tau_n) \to \tau''' \mathbin{@} \tau''$.
$\Sigma(a) = \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau'''$
Inversion on $\Sigma \vdash \texttt{ptr}(a) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau'''*$
$\mu(a) = v$ and $\Sigma \vdash v : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau'''$ $\mu : \Sigma$
$v = \texttt{func}(x_1, \ldots, x_n, e_b)$ Lemma 2
$\mu \mid K' \texttt{ F}', \texttt{call}(\texttt{ptr}(\texttt{a}), \cdot) \lhd (v_1, \ldots, v_n) \to$
$\mu \mid K'; F'; \cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n \rhd e_b$
Rule STEP-FINALIZECALL

B. Case CHECKFRAMETYPE-FRAMEEXPRET was used so $F' = F'', e'$
Progress holds via the same reasoning as in the previous case though only the ISPENDING-RETURN case occurs since there are no derivations of $\vdash e' \texttt{ returns}$ for the other cases.

(f) Case CHECKEXP-CALL was used so $e = \texttt{call}(e_f, e)$

$\bowtie = \rhd$ Inversion on $\bowtie \texttt{ call}(e_f, e_a) \texttt{ ok}$
$\mu \mid K'; F \rhd \texttt{call}(e_f, e) \to \mu \mid K'; F, \texttt{call}(\cdot, e) \rhd e_f$ Rule STEP-PUSHCALLFN

(g) Case CHECKEXP-IF was used so $e = \texttt{if}(e_c, e_t, e_f)$

$\bowtie \; = \; \rhd$ Inversion on $\bowtie \; \texttt{if}(e_c, e_t, e_f) \; \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{if}(e_c, e_t, e_f) \to \mu \mid K'; F, \texttt{if}(\cdot, e_t, e_f) \rhd e_c$ Rule STEP-PUSHIF

(h) Case CHECKEXP-DECL was used so $e = declx\tau_x e'$

$\bowtie \; = \; \rhd$ Inversion on $\bowtie \; \texttt{decl}(x, \tau_x, e') \; \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{decl}(x, \tau', e) \to \mu \mid K'; F, x : \tau' \rhd e$ Rule STEP-PUSHDECL

(i) Case CHECKEXP-ASSIGN was used so $e = \texttt{assign}(x, e')$

$\bowtie \; = \; \rhd$ Inversion on $\bowtie \; \texttt{assign}(x, e') \; \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{assign}(x, e) \to \mu \mid K'; F, \texttt{assign}(x, \cdot) \rhd e$ Rule STEP-PUSHASSIGN

(j) Case CHECKEXP-RETURN was used so $e = \texttt{return}(e')$

$\bowtie \; = \; \rhd$ Inversion on $\bowtie \; \texttt{return}(e') \; \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{return}(e) \to \mu \mid K'; F, \texttt{return}(\cdot) \rhd e$ Rule STEP-PUSHRET

(k) Case CHECKEXP-LOOP was used so $e = \texttt{loop}(e_c, e)$

$\bowtie \; = \; \rhd$ Inversion on $\bowtie \; \texttt{loop}(e_c, e') \; \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{loop}(e_c, e) \to \mu \mid K'; F, \texttt{loopctx}(e_c, e) \rhd \texttt{if}(e_c, e, \texttt{break})$
Rule STEP-LOOP

(l) Case CHECKEXP-BREAK was used so $e = \texttt{break}$

$\bowtie \; = \; \rhd$ Inversion on $\bowtie \; \texttt{break} \; \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{break} \to \mu \mid K'; F \lhd \texttt{break}$ Rule STEP-BREAK

(m) Case CHECKEXP-CONTINUE was used so $e = \texttt{continue}$

$\bowtie \; = \; \rhd$ Inversion on $\bowtie \; \texttt{continue} \; \texttt{ok}$
$\mu \mid K'; F \rhd \texttt{continue} \to \mu \mid K'; F \lhd \texttt{continue}$ Rule STEP-CONTINUE

(n) Case CHECKEXP-ALLOC was used so $e = \texttt{alloc}(\tau_a)$

$\bowtie = \triangleright$ Inversion on $\bowtie$ $\texttt{alloc}(\tau_a)$ $\texttt{ok}$

$\texttt{allocval}(\mu, a, \tau_a) = \mu'$ Lemma 6

$\mu \mid K'; F \triangleright \texttt{alloc}(\tau_a) \to \mu' \mid K'; F \triangleleft \texttt{ptr}(a)$ Rule STEP-ALLOC

(o) Case CHECKEXP-ALLOCARRAY was used so $e = \texttt{allocarray}(\tau_a, e')$

$\bowtie = \triangleright$ Inversion on $\bowtie$ $\texttt{allocarray}(\tau_a, e')$ $\texttt{ok}$

$\mu \mid K'; F \triangleright \texttt{allocarray}(\tau_a, e') \to \mu \mid K'; F, \texttt{allocarray}(\tau_a, \cdot) \triangleright e'$

Rule STEP-PUSHALLOCARRAY

4. Case CHECKSTATE-RETURNS was used
   By inversion on $\Sigma; \Gamma \vdash e$ $\texttt{returns}$ τ'' we know $\vdash e$ $\texttt{returns}$ so by case analysis, e is one of the following:
   - $\vdash \texttt{binop}(\texttt{seq}, e_1, e_2)$ $\texttt{returns}$
   - $\vdash \texttt{binop}(\texttt{seq}, e_1, e_2)$ $\texttt{returns}$
   - $\vdash \texttt{if}(e_c, e_t, e_f)$ $\texttt{returns}$
   - $\vdash \texttt{decl}(x, \tau, e)$ $\texttt{returns}$
   - $\vdash \texttt{return}(e)$ $\texttt{returns}$

   These cases are handled exactly the same as if the rule used in the derivation was CHECKSTATE-NORMAL.
5. Case CHECKSTATE-LOOPBRK was used so $K = K'; F$

   We now do case analysis on the structure of F:

   $F = F', x : \tau_x$

   $\mu \mid K'; F', x : \tau \triangleleft \texttt{break} \to \mu \mid K'; F' \triangleleft \texttt{break}$ Rule STEP-BREAKVAR

   $F = F', x : \tau_x = v$

   $\mu \mid K'; F', x : \tau_x = v \triangleleft \texttt{break} \to \mu \mid K'; F' \triangleleft \texttt{break}$ Rule STEP-BREAKVAL

   $F = F', \texttt{loopctx}(e_c, e)$

   $\mu \mid K'; F', \texttt{loopctx}(e_c, e') \triangleleft \texttt{break} \to \mu \mid K'; F' \triangleleft \texttt{nop}$ Rule STEP-BREAKLOOP

   $F = F', e$

   $\mu \mid K'; F', e' \triangleleft \texttt{break} \to \mu \mid K'; F' \triangleleft \texttt{break}$ Rule STEP-BREAKEXP

6. Case CHECKSTATE-LOOPCONT was used so $K = K'; F$

   We now do case analysis on the structure of F:

   $F = F', x : \tau_x$

   $$\mu \mid K'; F', x : \tau \lhd \texttt{continue} \to \mu \mid K'; F' \lhd \texttt{continue}$$ Rule STEP-CONTINUEVAR

   $F = F', x : \tau_x = v$

   $$\mu \mid K'; F', x : \tau_x = v \lhd \texttt{continue} \to \mu \mid K'; F' \lhd \texttt{continue}$$ Rule STEP-CONTINUEVAL

   $F = F', \texttt{loopctx}(e_c, e)$

   $$\mu \mid K'; F', \texttt{loopctx}(e_c, e') \lhd \texttt{continue} \to \mu \mid K'; F' \rhd \texttt{loop}(e_c, e')$$ Rule STEP-CONTINUELOOP

   $F = F', e$

   $$\mu \mid K'; F', e' \lhd \texttt{continue} \to \mu \mid K'; F' \lhd \texttt{continue}$$ Rule STEP-CONTINUEEXP

A.5.2 Preservation

Proof by induction on the derivation of $\mu \mid K; F \bowtie t \to \mu' \mid K' \bowtie' t'$. The cases where $\Sigma' \neq \Sigma$ are noted explicitly.

$$\frac{}{\mu \mid K; F \rhd v \to \mu \mid K; F \lhd v}$$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\Sigma; \Gamma \vdash v : \tau'$	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\texttt{assigned}(F) = A$	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\Gamma \vdash v : A \to A$	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\texttt{loopnest}(F) = L$	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$L \vdash v$ ok	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\Sigma; A \vdash F : \tau' \to \tau'''$	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\Sigma \vdash K : \tau'' \to \tau$	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash v$ canreturn τ''	Inversion on $\Sigma \vdash K; F \rhd v : \tau$
$\lhd v$ ok	Rule DIROK-RETURNING
$\Sigma \vdash K; F \lhd v : \tau$	Rule CHECKSTATE-NORMAL

$\mu \mid K; F, x : \tau_x = v, \ldots \rhd x \to \mu \mid K; F, x : \tau_x = v, \ldots \lhd v$

$\Sigma \vdash \texttt{ctx}(F, x : \tau_x = v, \ldots) = \Gamma$	Inversion on $\Sigma \vdash K; F \rhd x : \tau$
$\Sigma; \Gamma \vdash x : \tau'$	Inversion on $\Sigma \vdash K; F \rhd x : \tau$
$\texttt{assigned}(F, x : \tau_x = v, \ldots) = A$	Inversion on $\Sigma \vdash K; F \rhd x : \tau$
$\Gamma \vdash x : A \to A$	Inversion on $\Sigma \vdash K; F \rhd x : \tau$
$\texttt{loopnest}(F, x : \tau_x = v, \ldots) = L$	Inversion on $\Sigma \vdash K; F \rhd x : \tau$
$\Sigma; A \vdash F, x : \tau_x = v, \ldots : \tau' \to \tau''$	Inversion on $\Sigma \vdash K; F \rhd x : \tau$
$\Gamma = \Gamma', x : \tau'$	Inversion on $\Sigma; \Gamma \vdash x : \tau'$
$\Sigma; \Gamma \vdash v : \tau'$	Lemma 5
$L \vdash v$ ok	Rule CHECKLOOP-VALUE
$\Sigma \vdash K : \tau'' \to \tau$	Inversion on $\Sigma \vdash K; F \rhd x : \tau$
If $\tau' = \mathbf{cmd}$, $v = \texttt{nop}$	Lemma 2
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash v$ canreturn τ''	
	$\Sigma; \Gamma \vdash \texttt{nop}$ canreturn τ'' by rule ONLYRETURNS-NOP
$\lhd v$ ok	Rule DIROK-RETURNING
$\Sigma \vdash K; F, x : \tau_x = v, \ldots \lhd v : \tau$	Rule CHECKSTATE-NORMAL

$\mu \mid K; F \rhd \texttt{binop}(op, e_1, e_2) \to \mu \mid K; F, \texttt{binop}(op, \cdot, e_2) \rhd e_1$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$\texttt{assigned}(F) = A$	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$\Gamma \vdash \texttt{binop}(op, e_1, e_2) : A \to A''$	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$\texttt{loopnest}(F) = L$	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$L \vdash \texttt{binop}(op, e_1, e_2)$ ok	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2) : \tau'$	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$op : \tau_1 \times \tau_2 \to \tau'$ and $\Sigma; \Gamma \vdash e_1 : \tau_1$ and $\Sigma; \Gamma \vdash e_2 : \tau_2$	
	Inversion on $\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2) : \tau'$
$\Gamma \vdash e_1 : A \to A'$ and $\Gamma \vdash e_2 : A' \to A''$	Inversion on $\Gamma \vdash \texttt{binop}(op, e_1, e_2) : A \to A''$
$\Gamma \vdash \cdot : A' \to A'$	Rule CHECKASSIGN-HOLE
$\Gamma \vdash \texttt{binop}(op, \cdot, e_2) : A' \to A''$	Rule CHECKASSIGN-BINOP
$L \vdash e_1$ ok and $L \vdash e_2$ ok	Inversion on $\texttt{loopnest}(L) = \texttt{binop}(op, e_1, e_2)$
$L \vdash \cdot$ ok	Rule CHECKLOOP-VAR
$L \vdash \texttt{binop}(op, \cdot, e_2)$ ok	Rule CHECKLOOP-BINOP
If the rule used for $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$ was `checkstate-normal` then	
$\Sigma; A'' \vdash F : \tau' \to \tau'''$	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2)$ canreturn τ''	
	Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_1$ canreturn τ'' and $\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_2$ canreturn τ''	

Inversion on $\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2)\ \texttt{canreturn}\ \tau''$

If $\tau' = \mathbf{cmd}$ then $op = \texttt{seq}, \tau_1 = \mathbf{cmd}$, and $\tau_2 = \mathbf{cmd}$

Inversion on $op : \tau_1 \times \tau_2 \to \tau'$

$\tau' = \mathbf{cmd} \Leftrightarrow \tau_1 = \mathbf{cmd} \Leftrightarrow \tau_2 = \mathbf{cmd} \Leftrightarrow op = \texttt{seq}$

By case analysis of the rules for $op : \tau_1 \times \tau_2 \to \tau'$

Thus $\tau_1 = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_1\ \texttt{canreturn}\ \tau''$ and

$\tau_2 = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_2\ \texttt{canreturn}\ \tau''$

$\Sigma; \Gamma \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{pending}\ \tau_1 \to \tau' \mathbin{@} \tau''$ Rule ISPENDING-BINOPL

$\Sigma; A' \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, \texttt{binop}(op, \cdot, e_2)) = \Gamma$ Rule GETFRAMECONTEXT-EXP

$\texttt{assigned}(F, \texttt{binop}(op, \cdot, e_2)) = A$ Rule GETFRAMEASSIGNED-EXP

$\texttt{loopnest}(F, \texttt{binop}(op, \cdot, e_2)) = L$ Rule GETLOOPCONTEXT-EXP

$\rhd e_1\ \texttt{ok}$ Rule DIROK-PUSHING

$\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \rhd e_1 : \tau$ Rule CHECKSTATE-NORMAL

Otherwise the `checkstate-returns` rule was used.

$\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2)\ \texttt{returns}\ \tau''$

Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$

$\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2)\ \texttt{canreturn}\ \tau''$ and $\vdash \texttt{binop}(op, e_1, e_2)\ \texttt{returns}$ and $\tau''\ \texttt{small}$

Inversion on $\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2)\ \texttt{returns}\ \tau''$

$L = \texttt{notinloop}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$

$\tau' = \mathbf{cmd}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{binop}(op, e_1, e_2) : \tau$

$\tau' = \mathbf{cmd} \Leftrightarrow \tau_1 = \mathbf{cmd} \Leftrightarrow \tau_2 = \mathbf{cmd} \Leftrightarrow op = \texttt{seq}$

By case analysis of the rules for $op : \tau_1 \times \tau_2 \to \tau'$

Via case analysis on $\vdash \texttt{binop}(op, e_1, e_2)\ \texttt{returns}$, there are two possibilities.

Case $\vdash e_1\ \texttt{returns}$

$\Sigma; \Gamma \vdash e_1\ \texttt{canreturn}\ \tau''$

Inversion on $\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2)\ \texttt{canreturn}\ \tau''$

$\Sigma; \Gamma \vdash e_1\ \texttt{returns}\ \tau''$ Rule RETURNS-DEFAULT

$\Sigma \vdash \texttt{ctx}(F, \texttt{binop}(op, \cdot, e_2)) = \Gamma$ Rule GETFRAMECONTEXT-EXP

$\texttt{assigned}(F, \texttt{binop}(op, \cdot, e_2)) = A$ Rule GETFRAMEASSIGNED-EXP

$\texttt{loopnest}(F, \texttt{binop}(op, \cdot, e_2)) = \texttt{notinloop}$

Rule GETLOOPCONTEXT-EXP

$\rhd e_1\ \texttt{ok}$ Rule DIROK-PUSHING

$\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \rhd e_1 : \tau$ Rule CHECKSTATE-RETURNS

Case $\vdash e_2\ \texttt{returns}$

$\Sigma; \Gamma \vdash e_2\ \texttt{canreturn}\ \tau''$

Inversion on $\Sigma; \Gamma \vdash \texttt{binop}(op, e_1, e_2)\ \texttt{canreturn}\ \tau''$

$\tau_2 = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_2\ \texttt{canreturn}\ \tau''$ Weakening

$\Sigma; \Gamma \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{pending}\ \tau_1 \to \tau' \mathbin{@} \tau''$ Rule ISPENDING-BINOPL

$\vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{returns}$ Rule DOESRETURN-BINOPRHS

$\Sigma; \Gamma \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{returns}\ \tau''$ Rule RETURNS-DEFAULT

$\Sigma; A' \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$

Rule CHECKFRAMETYPE-FRAMEEXPRET

$\Sigma \vdash \texttt{ctx}(F, \texttt{binop}(op, \cdot, e_2)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F, \texttt{binop}(op, \cdot, e_2)) = A$ Rule GETFRAMEASSIGNED-EXP
$\texttt{loopnest}(F, \texttt{binop}(op, \cdot, e_2)) = L$ Rule GETLOOPCONTEXT-EXP
$\rhd e_1$ ok Rule DIROK-PUSHING
$\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \rhd e_1 : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F, \texttt{binop}(op, \cdot, e_2) \lhd v \to \mu \mid K; F, \texttt{binop}(op, v, \cdot) \rhd e_2$

$\Sigma \vdash \texttt{ctx}(F, \texttt{binop}(op, \cdot, e_2)) = \Gamma$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$
$\Sigma; \Gamma \vdash v : \tau_1$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$
$\texttt{assigned}(F, \texttt{binop}(op, \cdot, e_2)) = A$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$
$\Gamma \vdash v : A \to A$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$
$\texttt{loopnest}(F, \texttt{binop}(op, \cdot, e_2)) = L$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$
$L \vdash v$ ok Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$
$\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, \cdot, e_2) \lhd v : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\Gamma \vdash \texttt{binop}(op, \cdot, e_2) : A \to A'$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\Sigma; \Gamma \vdash \texttt{binop}(op, \cdot, e_2)$ pending $\tau_1 \to \tau' \mathbin{@} \tau''$
Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$op : \tau_1 \times \tau_2 \to \tau'$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\tau_2 = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_2$ canreturn τ''
Inversion on $\Sigma; \Gamma \vdash \texttt{binop}(op, \cdot, e_2)$ pending $\tau_1 \to \tau' \mathbin{@} \tau''$
$\Sigma; \Gamma \vdash e_2 : \tau_2$ Inversion on $\Sigma; \Gamma \vdash \texttt{binop}(op, \cdot, e_2)$ pending $\tau_1 \to \tau' \mathbin{@} \tau''$
$\Gamma \vdash e_2 : A \to A'$ Inversion on $\Gamma \vdash \texttt{binop}(op, \cdot, e_2) : A \to A$

There are two possible rules used for the derivation of $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
Case CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma; A' \vdash F : \tau' \to \tau''$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\Gamma \vdash \cdot : A' \to A'$ Rule CHECKASSIGN-HOLE
$\Gamma \vdash v : A' \to A'$ Rule CHECKASSIGN-VALUE
$\Gamma \vdash \texttt{binop}(op, v, \cdot) : A' \to A'$ Rule CHECKASSIGN-BINOP
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$L \vdash \cdot$ ok Rule CHECKLOOP-VAR
$L \vdash \texttt{binop}(op, v, \cdot)$ ok Rule CHECKLOOP-BINOP
$\Sigma \vdash v : \tau_1$ Inversion on $\Sigma; \Gamma \vdash v : \tau_1$
$\Sigma; \Gamma \vdash \texttt{binop}(op, v, \cdot)$ pending $\tau_2 \to \tau' \mathbin{@} \tau''$ Rule ISPENDING-BINOPR
$\Sigma; A' \vdash F, \texttt{binop}(op, v, \cdot) : \tau_2 \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, \texttt{binop}(op, v, \cdot)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F) = A$ Inversion on $\texttt{assigned}(F, \texttt{binop}(op, \cdot, e_2)) = A$

$\texttt{assigned}(F, \texttt{binop}(op, v, \cdot)) = A$ Rule GETFRAMEASSIGNED-EXP
$\texttt{loopnest}(F, \texttt{binop}(op, v, \cdot)) = L$ Rule GETLOOPCONTEXT-EXP
$L \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{ok}$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$L \vdash e_2\ \texttt{ok}$ Inversion on $L \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{ok}$
$\rhd e_2\ \texttt{ok}$ Rule DIROK-PUSHING
$\Sigma \vdash K; F, \texttt{binop}(op, v, \cdot) \rhd e_2 : \tau$ Rule CHECKSTATE-NORMAL

Otherwise CHECKFRAMETYPE-FRAMEEXPRET was used

$\vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{returns}$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\texttt{loopnest}(F) = \texttt{notinloop}$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\texttt{notinloop} \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{ok}$
Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\tau' = \mathbf{cmd}$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, \cdot, e_2) : \tau_1 \to \tau''$
$\tau' = \mathbf{cmd} \Leftrightarrow \tau_2 = \mathbf{cmd} \Leftrightarrow op = \texttt{seq}$
$\Sigma \vdash \texttt{ctx}(F, \texttt{binop}(op, v, \cdot)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F, \texttt{binop}(op, v, \cdot)) = A$ Rule GETFRAMEASSIGNED-EXP
$\texttt{loopnest}(F, \texttt{binop}(op, v, \cdot)) = \texttt{notinloop}$ Rule GETLOOPCONTEXT-EXP
$\texttt{notinloop} \vdash e_2\ \texttt{ok}$ Inversion on $\texttt{notinloop} \vdash \texttt{binop}(op, \cdot, e_2)\ \texttt{ok}$
$\rhd e_2\ \texttt{ok}$ Rule DIROK-PUSHING
$\Sigma; \Gamma \vdash e_2\ \texttt{canreturn}\ \tau''$ Modus ponens
$\Sigma; \Gamma \vdash e_2\ \texttt{returns}\ \tau''$ Rule RETURNS-ONLY
$\Sigma \vdash K; F, \texttt{binop}(op, v, \cdot) \rhd e_2 : \tau$ Rule CHECKSTATE-RETURNS

$$\frac{\mu\ \mid op(v_1, v_2) \to \mu'\ \mid t}{\mu\ \mid K; F, \texttt{binop}(op, v_1, \cdot) \lhd v_2 \to \mu'\ \mid K; F \lhd t}$$

$\Sigma \vdash \texttt{ctx}(F, \texttt{binop}(op, v_1, \cdot)) = \Gamma$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, v_1, \cdot) \rhd v_2 : \tau$
$\Sigma; \Gamma \vdash v_2 : \tau_2$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, v_1, \cdot) \rhd v_2 : \tau$
$\texttt{assigned}(F, \texttt{binop}(op, v_1, \cdot)) = A$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, v_1, \cdot) \rhd v_2 : \tau$
$\Gamma \vdash v_2 : A \to A$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, v_1, \cdot) \rhd v_2 : \tau$
$\texttt{loopnest}(\texttt{binop}(op, v_1, \cdot)) = L$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, v_1, \cdot) \rhd v_2 : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, v_1, \cdot) \rhd v_2 : \tau$
$\Sigma; A \vdash F, \texttt{binop}(op, v_1, \cdot) : \tau_2 \to \tau'$ Inversion on $\Sigma \vdash K; F, \texttt{binop}(op, v_1, \cdot) \rhd v_2 : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, v_1, \cdot) : \tau_2 \to \tau'$
$\texttt{assigned}(F) = A$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, v_1, \cdot) : \tau_2 \to \tau'$
$\Gamma \vdash \texttt{binop}(op, v, \cdot) : A \to A$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, v_1, \cdot) : \tau_2 \to \tau'$
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, v_1, \cdot) : \tau_2 \to \tau'$
$\Sigma; A \vdash F : \tau' \to \tau''$ Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, v_1, \cdot) : \tau_2 \to \tau'$
$\Sigma; \Gamma \vdash \texttt{binop}(op, v_1, \cdot)\ \texttt{pending}\ \tau_2 \to \tau' \mathbin{@} \tau''$
Inversion on $\Sigma; A \vdash F, \texttt{binop}(op, v_1, \cdot) : \tau_2 \to \tau'$
$\Sigma; \Gamma \vdash v : \tau_1$ and $op : \tau_1 \times \tau_2 \to \tau'$
Inversion on $\Sigma; \Gamma \vdash \texttt{binop}(op, v_1, \cdot)\ \texttt{pending}\ \tau_2 \to \tau' \mathbin{@} \tau''$
$\mu\ \mid op(v_1, v_2) \to \mu'\ \mid t$ Inversion on $\mu\ \mid K; F, \texttt{binop}(op, v_1, \cdot) \lhd v_2 \to \mu'\ \mid K; F \lhd t$

$\mu' : \Sigma$ and either $t = \texttt{exn}$ or else $t = v'$ and $\Sigma \vdash v' : \tau'$ — Lemma 13
If $t = \texttt{exn}$ then $\Sigma \vdash K \lhd \texttt{exn} : \tau$ — Rule CHECKSTATE-EXN
Otherwise, $t = v'$ and $\Sigma \vdash v' : \tau'$
$\Sigma; \Gamma \vdash v' : \tau'$ — Rule CHECKEXP-VALUE
$\Gamma \vdash v' : A \to A$ — Rule CHECKASSIGN-VALUE
$L \vdash v'\ \texttt{ok}$ — Rule CHECKLOOP-VALUE
$\lhd v\ \texttt{ok}$ — Rule DIROK-RETURNING
If $\tau' = \mathbf{cmd}$ then $v' = \texttt{nop}$ — Lemma 2
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash v'\ \texttt{canreturn}\ \tau''$ — $\Sigma; \Gamma \vdash \texttt{nop}\ \texttt{canreturn}\ \tau''$
$\Sigma \vdash K; F \lhd v' : \tau$ — Rule CHECKSTATE-NORMAL

$\mu\ \mid K; F \rhd \texttt{monop}(op, e) \to \mu\ \mid K; F, \texttt{monop}(op, \cdot) \rhd e$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$
$\Sigma; \Gamma \vdash \texttt{monop}(op, e) : \tau'''$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$
$\texttt{assigned}(F) = A$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$
$\Gamma \vdash \texttt{monop}(op, e) : A \to A$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$
$\texttt{loopnest}(F) = L$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$
$L \vdash \texttt{monop}(op, e)\ \texttt{ok}$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$
$\Sigma; A \vdash F : \tau''' \to \tau''$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{monop}(op, e) : \tau$

$\Gamma \vdash \cdot : A \to A$ — Rule CHECKASSIGN-HOLE
$\Gamma \vdash \texttt{monop}(op, \cdot) : A \to A$ — Rule CHECKASSIGN-MONOP
$L \vdash \cdot\ \texttt{ok}$ — Rule CHECKLOOP-VAR
$L \vdash \texttt{monop}(op, \cdot)\ \texttt{ok}$ — Rule CHECKLOOP-MONOP
$op : \tau' \to \tau'''$ — Inversion on $\Sigma; \Gamma \vdash \texttt{monop}(op, e) : \tau'''$
$\Sigma; \Gamma \vdash \texttt{monop}(op, \cdot)\ \texttt{pending}\ \tau' \to \tau''' \mathbin{@} \tau''$ — Rule ISPENDING-MONOP
$\Sigma; A \vdash F, \texttt{monop}(op, \cdot) : \tau' \to \tau'''$ — Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, \texttt{monop}(op, \cdot)) = \Gamma$ — Rule GETFRAMECONTEXT-EXP
$\Sigma; \Gamma \vdash e : \tau'$ — Inversion on $\Sigma; \Gamma \vdash \texttt{monop}(op, e) : \tau'''$
$\texttt{assigned}(F, \texttt{monop}(op, \cdot)) = A$ — Rule GETFRAMEASSIGNED-EXP
$\Gamma \vdash e : A \to A$ — Inversion on $\Gamma \vdash \texttt{monop}(op, e) : A \to A$
$L \vdash e\ \texttt{ok}$ — Inversion on $L \vdash \texttt{monop}(op, e)\ \texttt{ok}$
$\rhd e\ \texttt{ok}$ — Rule DIROK-PUSHING
$\tau' \neq \mathbf{cmd}$ — Inversion on $op : \tau' \to \tau'''$
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \tau'\ \texttt{canreturn}\ \tau''$ — Vacuously true
$\Sigma \vdash K; F, \texttt{monop}(op, \cdot) \rhd e : \tau$ — Rule CHECKSTATE-NORMAL

$$\frac{\mu\ \mid op(v) \to t}{\mu\ \mid K; F, \texttt{monop}(op, \cdot) \lhd v \to \mu\ \mid K; F \lhd t}$$

$\Sigma \vdash \texttt{ctx}(F, \texttt{monop}(op, \cdot)) = \Gamma$ Inversion on $\Sigma \vdash K; F, \texttt{monop}(op, \cdot) \lhd v : \tau$
$\Sigma; \Gamma \vdash v : \tau'$ Inversion on $\Sigma \vdash K; F, \texttt{monop}(op, \cdot) \lhd v : \tau$
$\texttt{assigned}(F, \texttt{monop}(op, \cdot)) = A$ Inversion on $\Sigma \vdash K; F, \texttt{monop}(op, \cdot) \lhd v : \tau$
$\Gamma \vdash v : A \to A$ Inversion on $\Sigma \vdash K; F, \texttt{monop}(op, \cdot) \lhd v : \tau$
$\Sigma; A \vdash F, \texttt{monop}(op, \cdot) : \tau' \to \tau''$ Inversion on $\Sigma \vdash K; F, \texttt{monop}(op, \cdot) \lhd v : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, \texttt{monop}(op, \cdot) \lhd v : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash \texttt{ctx}(F, \texttt{monop}(op, \cdot)) = \Gamma$
$\Gamma \vdash \texttt{monop}(op, \cdot) : A \to A$ Inversion on $\Sigma; A \vdash F, \texttt{monop}(op, \cdot) : \tau' \to \tau''$
$\Sigma; \Gamma \vdash \texttt{monop}(op, \cdot)\ \texttt{pending}\ \tau' \to \tau''' \mathbin{@} \tau''$
Inversion on $\Sigma; A \vdash F, \texttt{monop}(op, \cdot) : \tau' \to \tau''$
$\Sigma; A \vdash F : \tau''' \to \tau''$ Inversion on $\Sigma; A \vdash F, \texttt{monop}(op, \cdot) : \tau' \to \tau''$
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma; A \vdash F, \texttt{monop}(op, \cdot) : \tau' \to \tau''$
$op : \tau' \to \tau'''$ Inversion on $\Sigma; \Gamma \vdash \texttt{monop}(op, \cdot)\ \texttt{pending}\ \tau' \to \tau''' \mathbin{@} \tau''$
Either $t = \texttt{exn}$ or $t = v'$ where $\Sigma \vdash v' : \tau'''$ Lemma 14
If $t = \texttt{exn}$
Trivially $\Sigma \vdash K \lhd \texttt{exn} : \tau$
Otherwise $t = v'$ where $\Sigma \vdash v' : \tau'''$
$\Sigma; \Gamma \vdash v' : \tau'''$ Rule CHECKEXP-VALUE
$\texttt{assigned}(F) = A$ Inversion on $\texttt{assigned}(F, \texttt{monop}(op, \cdot)) = A$
$\Gamma \vdash v' : A \to A$ Rule CHECKASSIGN-VALUE
$L \vdash v'\ \texttt{ok}$ Rule CHECKLOOP-VALUE
$\lhd v\ \texttt{ok}$ Rule DIROK-RETURNING
If $\tau''' = \mathbf{cmd}$ then $v' = \texttt{nop}$ Lemma 2
$\tau''' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash v\ \texttt{canreturn}\ \tau''$ $\Sigma; \Gamma \vdash \texttt{nop}\ \texttt{canreturn}\ \tau''$
$\Sigma \vdash K; F \lhd v' : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F \rhd \texttt{call}(e_f, e) \to \mu \mid K; F, \texttt{call}(\cdot, e) \rhd e_f$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$
$\Sigma; \Gamma \vdash \texttt{call}(e_f, e) : \tau_r$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$
$\texttt{assigned}(F) = A$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$
$\Gamma \vdash \texttt{call}(e_f, e) : A \to A$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$
$L \vdash \texttt{call}(e_f, e)\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$
$\Sigma; A \vdash F : \tau_r \to \tau''$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F \rhd \texttt{call}(e_f, e) : \tau$

$\Gamma \vdash e : A \to A$ Inversion on $\Gamma \vdash \texttt{call}(e_f, e) : A \to A$
$\Gamma \vdash \cdot : A \to A$ Rule CHECKASSIGN-HOLE
$\Gamma \vdash \texttt{call}(\cdot, e) : A \to A$ Rule CHECKASSIGN-CALL

$L \vdash e\ \texttt{ok}$ Inversion on $L \vdash \texttt{call}(e_f, e)\ \texttt{ok}$
$L \vdash \cdot\ \texttt{ok}$ Rule CHECKLOOP-VAR
$L \vdash \texttt{call}(\cdot, e)\ \texttt{ok}$ Rule CHECKLOOP-CALL
$\Sigma; \Gamma \vdash e : \texttt{tuple}(\tau_1, \ldots, \tau_n)$ Inversion on $\Sigma; \Gamma \vdash \texttt{call}(e_f, e) : \tau_r$
$\Sigma; \Gamma \vdash \texttt{call}(\cdot, e)\ \texttt{pending}\ (\texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r)* \to \tau_r\ @\ \tau''$ Rule ISPENDING-CALLF
$\Sigma; A \vdash F, \texttt{call}(\cdot, e) : (\texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r)* \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, \texttt{call}(\cdot, e)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\Sigma; \Gamma \vdash e_f : (\texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r)*$ Inversion on $\Sigma; \Gamma \vdash \texttt{call}(e_f, e) : \tau_r$
$\texttt{assigned}(F, \texttt{call}(\cdot, e)) = A$ Rule GETFRAMEASSIGNED-EXP
$\Gamma \vdash e_f : A \to A$ Inversion on $\Gamma \vdash \texttt{call}(e_f, e) : A \to A$
$\texttt{loopnest}(F, \texttt{call}(\cdot, e)) = L$ Rule GETLOOPCONTEXT-EXP
$L \vdash e_f\ \texttt{ok}$ Inversion on $L \vdash \texttt{call}(e_f)e\ \texttt{ok}$
$\rhd e_f\ \texttt{ok}$ Rule DIROK-PUSHING
$(\texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r)* = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_f\ \texttt{canreturn}\ \tau''$ Vacuously true
$\Sigma \vdash K; F, \texttt{call}(\cdot, e) \rhd e_f : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F, \texttt{call}(\cdot, e) \lhd v \to \mu \mid K; F, \texttt{call}(v, \cdot) \rhd e$

$\Sigma \vdash \texttt{ctx}(F, \texttt{call}(\cdot, e)) = \Gamma$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\cdot, e) \lhd v : \tau$
$\texttt{assigned}(F, \texttt{call}(\cdot, e)) = A$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\cdot, e) \lhd v : \tau$
$\Gamma \vdash v : A \to A$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\cdot, e) \lhd v : \tau$
$\Sigma; A \vdash F, \texttt{call}(\cdot, e) : (\tau_a \to \tau_r)* \to \tau_r$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\cdot, e) \lhd v : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\cdot, e) \lhd v : \tau$
$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma; A \vdash F, \texttt{call}(\cdot, e) : (\tau_a \to \tau_r)* \to \tau_r$
$\Sigma; \Gamma \vdash \texttt{call}(\cdot, e)\ \texttt{pending}\ (\tau_a \to \tau_r)* \to \tau_r\ @\ \tau''$ Inversion on $\Sigma; A \vdash F, \texttt{call}(\cdot, e) : (\tau_a \to \tau_r)* \to \tau_r$
$\Sigma; \Gamma \vdash v : (\tau_a \to \tau_r)*$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\cdot, e) \lhd v : \tau$
$v = \texttt{ptr}(a)$ where $\Sigma(a) = \tau_a \to \tau_r$ Lemma 2
$\Sigma; \Gamma \vdash \texttt{ptr}(a) : (\tau_a \to \tau_r)*$ Rule CHECKEXP-VAR
$\Sigma; \Gamma \vdash \texttt{call}(\texttt{ptr}(a), \cdot)\ \texttt{pending}\ \tau_a \to \tau_r\ @\ \tau''$ Rule ISPENDING-CALLA
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma; A \vdash F, \texttt{call}(\cdot, e) : (\tau_a \to \tau_r)* \to \tau_r$
$L \vdash \texttt{call}(v, \cdot)\ \texttt{ok}$ Rule CHECKLOOP-CALL
$\Gamma \vdash v : A \to A$ Rule CHECKASSIGN-VALUE
$\Gamma \vdash \cdot : A \to A$ Rule CHECKASSIGN-HOLE
$\Gamma \vdash \texttt{call}(v, \cdot) : A \to A$ Rule CHECKASSIGN-CALL
$\Sigma; A \vdash F, \texttt{call}(v, \cdot) : \tau_a \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET
$\Sigma \vdash \texttt{ctx}(F, \texttt{call}(v, \cdot)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\Sigma; \Gamma \vdash e : \tau_a$ Inversion on $\Sigma; \Gamma \vdash \texttt{call}(\cdot, e)\ \texttt{pending}\ (\tau_a \to \tau_r)* \to \tau_r\ @\ \tau''$
$\texttt{assigned}(F) = A$ Inversion on $\texttt{assigned}(F, \texttt{call}(\cdot, e)) = A$

$\texttt{assigned}(F, \texttt{call}(v, \cdot)) = A$ Rule GETFRAMEASSIGNED-EXP
$\Gamma \vdash \texttt{call}(\cdot, e) : A \to A$ Inversion on $\Sigma; A \vdash F, \texttt{call}(\cdot, e) : (\tau_a \to \tau_r)* \to \tau_r$
$\Gamma \vdash e : A \to A$ Inversion on $\Gamma \vdash \texttt{call}(\cdot, e) : A \to A$
$\texttt{loopnest}(F, \texttt{call}(v, \cdot)) = L$ Rule GETLOOPCONTEXT-EXP
$L \vdash \texttt{call}(\cdot, e)\ \texttt{ok}$ Inversion on $\Sigma; A \vdash F, \texttt{call}(\cdot, e) : (\tau_a \to \tau_r)* \to \tau_r$
$L \vdash e\ \texttt{ok}$ Inversion on $L \vdash \texttt{call}(\cdot, e)\ \texttt{ok}$
$\rhd e\ \texttt{ok}$ Rule DIROK-PUSHING
$\mu(a) = v_f$ where $\Sigma \vdash v_f : \tau_a \to \tau_r$ $\mu : \Sigma$
$v_f = \texttt{func}(x_1, \ldots, x_n, e')$ Lemma 2
$\tau_a = \texttt{tuple}(\tau_1, \ldots, \tau_n)$ Inversion on $\Sigma \vdash v_f : \tau_a \to \tau_r$
$\tau_a = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e\ \texttt{canreturn}\ \tau''$ Vacuously true
$\Sigma \vdash K; F, \texttt{call}(v, \cdot) \rhd e : \tau$ Rule CHECKSTATE-NORMAL

$$\frac{\mu(a) = \texttt{func}(x_1, \ldots, x_n, e)}{\mu \mid K; F, \texttt{call}(\texttt{ptr}(a), \cdot) \lhd (v_1, \ldots, v_n) \to \mu \mid K; F; \cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n \rhd e}$$

$\Sigma; \Gamma \vdash (v_1, \ldots, v_n) : \texttt{tuple}(\tau_1, \ldots, \tau_n)$
Inversion on $\Sigma \vdash K; F, \texttt{call}(\texttt{ptr}(a), \cdot) \lhd (v_1, \ldots, v_n) : \tau$
$\texttt{assigned}(F, \texttt{call}(\texttt{ptr}(a), \cdot)) = A$
Inversion on $\Sigma \vdash K; F, \texttt{call}(\texttt{ptr}(a), \cdot) \lhd (v_1, \ldots, v_n) : \tau$
$\Gamma \vdash (v_1, \ldots, v_n) : A \to A$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\texttt{ptr}(a), \cdot) \lhd (v_1, \ldots, v_n) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, \texttt{call}(\texttt{ptr}(a), \cdot) \lhd (v_1, \ldots, v_n) : \tau$
$\Sigma; A \vdash F, \texttt{call}(\texttt{ptr}(a), \cdot) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau''$
Inversion on $\Sigma \vdash K; F, \texttt{call}(\texttt{ptr}(a), \cdot) \lhd (v_1, \ldots, v_n) : \tau$
$\Gamma \vdash \texttt{call}(\texttt{ptr}(a), \cdot) : A \to A$
Inversion on $\Sigma; A \vdash F, \texttt{call}(\texttt{ptr}(a), \cdot) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau''$
$\Sigma; \Gamma \vdash \texttt{call}(\texttt{ptr}(a), \cdot)\ \texttt{pending}\ \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r\ @\ \tau''$
Inversion on $\Sigma; A \vdash F, \texttt{call}(\texttt{ptr}(a), \cdot) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau''$
$\Sigma; \Gamma \vdash \texttt{ptr}(a) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r*$
Inversion on $\Sigma; \Gamma \vdash \texttt{call}(\texttt{ptr}(a), \cdot)\ \texttt{pending}\ \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r\ @\ \tau''$
$\Sigma \vdash \texttt{func}(x_1, \ldots, x_n, e) : \texttt{tuple}(\tau_1, \ldots \tau_n) \to \tau_r$
Inversion on $\Sigma; \Gamma \vdash \texttt{ptr}(a) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau_r*$
$\Sigma; A \vdash F : \tau_r \to \tau''$ Inversion on $\Sigma; A \vdash F, \texttt{call}(\texttt{ptr}(a), \cdot) : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau''$
$\texttt{assigned}(F) = A$ Inversion on $\texttt{assigned}(F, \texttt{call}(\texttt{ptr}(a), \cdot)) = A$
$\Sigma \vdash K; F : \tau_r \to \tau$ Rule CHECKSTACK-NONEMPTY
$\Sigma \vdash \texttt{ctx}(\cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n) = \Gamma'$ Lemma 15
$\texttt{assigned}(\cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n) = A'$ Lemma 16
$\Sigma; \Gamma \vdash e : \mathbf{cmd}$ Inversion on $\Sigma \vdash \texttt{func}(x_1, \ldots, x_n, e) : \texttt{tuple}(\tau_1, \ldots \tau_n) \to \tau_r$
$\Gamma \vdash e : A \to A'$ Inversion on $\Sigma \vdash \texttt{func}(x_1, \ldots, x_n, e) : \texttt{tuple}(\tau_1, \ldots \tau_n) \to \tau_r$
$\texttt{loopnest}(F) = \texttt{notinloop}$ By construction
$\texttt{notinloop} \vdash e\ \texttt{ok}$ Inversion on $\Sigma \vdash \texttt{func}(x_1, \ldots, x_n, e) : \texttt{tuple}(\tau_1, \ldots \tau_n) \to \tau_r$
$\Sigma; \Gamma \vdash e\ \texttt{returns}\ \tau_r$ Inversion on $\Sigma \vdash \texttt{func}(x_1, \ldots, x_n, e) : \texttt{tuple}(\tau_1, \ldots \tau_n) \to \tau_r$
$\rhd e\ \texttt{ok}$ Rule DIROK-PUSHING

$\Sigma \vdash K; F; \cdot, x_1 : \tau_1 = v_1, \ldots, x_n : \tau_n = v_n \rhd e : \tau$ Rule CHECKSTATE-RETURNS

$\mu \mid K; F, \texttt{call}(\texttt{ptr}(\textbf{null}), \cdot) \lhd (v_1, \ldots, v_n) \to \mu \mid K; F \lhd \texttt{exn}$

$\Sigma \vdash K; F \lhd \texttt{exn} : \tau$ Rule CHECKSTATE-EXN

$\mu \mid K; F \rhd () \to \mu \mid K; F \lhd ()$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$\Sigma; \Gamma \vdash () : \texttt{tuple}()$	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$\texttt{assigned}(F) = A$	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$\Gamma \vdash () : A \to A$	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$\Sigma; A \vdash F : \texttt{tuple}() \to \tau''$	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$\Sigma \vdash K : \tau'' \to \tau$	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$\texttt{loopnest}(F) = L$	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$L \vdash ()$ ok	Inversion on $\Sigma \vdash K; F \rhd () : \tau$
$\texttt{tuple}() = \textbf{cmd} \Rightarrow \Sigma; \Gamma \vdash ()$ canreturn τ''	Vacuously true
$\lhd ()$ ok	Rule DIROK-TUPLE
$\Sigma \vdash K; F \lhd () : \tau$	Rule CHECKSTATE-NORMAL

$\mu \mid K; F \rhd (e_1, \ldots) \to \mu \mid K; F, (\cdot, \ldots) \rhd e_1$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$
$\Sigma; \Gamma \vdash (e_1, \ldots) : \texttt{tuple}(\tau_1, \ldots)$	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$
$\texttt{assigned}(F) = A$	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$
$\Gamma \vdash (e_1, \ldots) : A \to A$	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$
$\texttt{loopnest}(F) = L$	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$
$L \vdash (e_1, \ldots)$ ok	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$
$\Sigma; A \vdash F : \texttt{tuple}(\tau_1, \ldots) \to \tau''$	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$	Inversion on $\Sigma \vdash K; F \rhd (e_1, \ldots) : \tau$

$\Sigma; \Gamma \vdash e_1 : \tau_1 \ldots$	Inversion on $\Sigma; \Gamma \vdash (e_1, \ldots) : \texttt{tuple}(\tau_1, \ldots)$
$\Gamma \vdash e_1 : A \to A \ldots$	Inversion on $\Gamma \vdash (e_1, \ldots) : A \to A$
$L \vdash e_1$ ok ...	Inversion on $L \vdash (e_1, \ldots)$ ok
τ_1 small ...	Inversion on $\Sigma; \Gamma \vdash (e_1, \ldots) : \texttt{tuple}(\tau_1, \ldots)$

$\Gamma \vdash \cdot : A \to A$	Rule CHECKASSIGN-HOLE
$\Gamma \vdash (\cdot, \ldots) : A \to A$	Rule CHECKASSIGN-TUPLE
$L \vdash \cdot$ ok	Rule CHECKLOOP-VAR
$L \vdash (\cdot, \ldots)$ ok	Rule CHECKLOOP-TUPLE

$\Sigma; \Gamma \vdash (\cdot, \ldots)\ \texttt{pending}\ \tau_1 \to \texttt{tuple}(\tau_1, \ldots) \mathbin{@} \tau''$ Rule ISPENDING-TUPLE
$\Sigma; A \vdash F, (\cdot, \ldots) : \tau_1 \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, (\cdot, \ldots)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F, (\cdot, \ldots)) = A$ Rule GETFRAMEASSIGNED-EXP
$\texttt{loopnest}(F, (\cdot, \ldots)) = L$ Rule GETLOOPCONTEXT-EXP
$\rhd e_1\ \texttt{ok}$ Rule DIROK-PUSHING
$\tau_1 \neq \mathbf{cmd}$ No possible derivation of $\mathbf{cmd}\ \texttt{small}$
$\tau_1 = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_1\ \texttt{canreturn}\ \tau''$ Vacuously true
$\Sigma \vdash K; F, (\cdot, \ldots) \rhd e_1 : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F, (\ldots, \cdot, e_i, \ldots) \lhd v \to \mu \mid K; F, (\ldots, v, \cdot, \ldots) \rhd e_i$

$\Sigma \vdash \texttt{ctx}(F, (\ldots, \cdot, e_i, \ldots)) = \Gamma$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$
$\Sigma; \Gamma \vdash v : \tau_{i-1}$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$
$\texttt{assigned}(F, (\ldots, \cdot, e_i, \ldots)) = A$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$
$\Gamma \vdash v : A \to A$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$
$\texttt{loopnest}(F, (\ldots, \cdot, e_i, \ldots)) = L$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$
$L \vdash v\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$
$\Sigma; A \vdash F, (\ldots, \cdot, e_i, \ldots) : \tau_{i-1} \to \tau''$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot, e_i, \ldots) \lhd v : \tau$

$\Gamma \vdash (\ldots, \cdot, e_i, \ldots) : A \to A$ Inversion $\Sigma; A \vdash F, (\ldots, \cdot, e_i, \ldots) : \tau_{i-1} \to \tau''$
$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion $\Sigma; A \vdash F, (\ldots, \cdot, e_i, \ldots) : \tau_{i-1} \to \tau''$
$\texttt{loopnest}(F) = L$ Inversion $\Sigma; A \vdash F, (\ldots, \cdot, e_i, \ldots) : \tau_{i-1} \to \tau''$
$L \vdash (\ldots, \cdot, e_i, \ldots)\ \texttt{ok}$ Inversion $\Sigma; A \vdash F, (\ldots, \cdot, e_i, \ldots) : \tau_{i-1} \to \tau''$
$\Sigma; A \vdash F : \texttt{tuple}(\tau_1, \ldots, \tau_n) \to \tau''$ Inversion $\Sigma; A \vdash F, (\ldots, \cdot, e_i, \ldots) : \tau_{i-1} \to \tau''$
$\ldots\ \Gamma \vdash \cdot : A \to A\ \Gamma \vdash e_i : A \to A \ldots$ Inversion on $\Gamma \vdash (\ldots, \cdot, e_i, \ldots) : A \to A$
$\Sigma; \Gamma \vdash (\ldots, \cdot, e_i, \ldots)\ \texttt{pending}\ \tau_{i-1} \to \texttt{tuple}(\tau_1, \ldots, \tau_n) \mathbin{@} \tau''$
Inversion $\Sigma; A \vdash F, (\ldots, \cdot, e_i, \ldots) : \tau_{i-1} \to \tau''$
$\ldots\ \Sigma; \Gamma \vdash e_i : \tau_i$
Inversion on $\Sigma; \Gamma \vdash (\ldots, \cdot, e_i, \ldots)\ \texttt{pending}\ \tau_{i-1} \to \texttt{tuple}(\tau_1, \ldots, \tau_n) \mathbin{@} \tau''$
$\tau_1\ \texttt{small}\ \ldots \tau_n\ \texttt{small}$
Inversion on $\Sigma; \Gamma \vdash (\ldots, \cdot, e_i, \ldots)\ \texttt{pending}\ \tau_{i-1} \to \texttt{tuple}(\tau_1, \ldots, \tau_n) \mathbin{@} \tau''$
$\ldots\ L \vdash \cdot\ \texttt{ok}\ \ L \vdash e_i\ \texttt{ok}\ \ldots$ Inversion on $L \vdash (\ldots, \cdot, e_i, \ldots)\ \texttt{ok}$

$\Gamma \vdash (\ldots, v, \cdot, \ldots) : A \to A$ Rule CHECKASSIGN-TUPLE
$L \vdash (\ldots, v, \cdot, \ldots)\ \texttt{ok}$ Rule CHECKLOOP-TUPLE
$\Sigma; \Gamma \vdash (\ldots, v, \cdot, \ldots)\ \texttt{pending}\ \tau_i \to \texttt{tuple}(\tau_1, \ldots, \tau_n) \mathbin{@} \tau''$ Rule ISPENDING-TUPLE
$\Sigma; A \vdash F, (\ldots, v, \cdot, \ldots) : \tau_i \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, (\ldots, v, \cdot, \ldots)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F, (\ldots, v, \cdot, \ldots)) = A$ Rule GETFRAMEASSIGNED-EXP

$\texttt{loopnest}(F, (\ldots, v, \cdot, \ldots)) = L$ Rule GETLOOPCONTEXT-EXP

$\triangleright\ e_i\ \texttt{ok}$ Rule DIROK-PUSHING

$\tau_i \neq \textbf{cmd}$ No possible derivation of **cmd** `small`

$\tau_i = \textbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_i\ \texttt{canreturn}\ \tau''$ Vacuously true

$\Sigma \vdash K; F, (\ldots, v, \cdot, \ldots) \triangleright e_i : \tau$ Rule CHECKSTATE-NORMAL

$\mu\ \mid K; F, (\ldots, \cdot) \triangleleft v \rightarrow \mu\ \mid K; F \triangleleft (\ldots, v)$

$\Sigma \vdash \texttt{ctx}(F, (\ldots, \cdot)) = \Gamma$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$\Sigma; \Gamma \vdash v : \tau_n$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$\texttt{assigned}(F, (\ldots, \cdot)) = A$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$\Gamma \vdash v : A \rightarrow A$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$\texttt{loopnest}(F, (\ldots, \cdot)) = L$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$L \vdash v\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$\Sigma; A \vdash F, (\ldots, \cdot) : \tau_n \rightarrow \tau''$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$\Sigma \vdash K : \tau'' \rightarrow \tau$ Inversion on $\Sigma \vdash K; F, (\ldots, \cdot) \triangleleft v : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash \texttt{ctx}(F, (\ldots, \cdot)) = \Gamma$

$\Gamma \vdash (\ldots, \cdot) : A \rightarrow A$ Inversion on $\Sigma; A \vdash F, (\ldots, \cdot) : \tau_n \rightarrow \tau''$

$\texttt{loopnest}(F) = L$ Inversion on $\Sigma; A \vdash F, (\ldots, \cdot) : \tau_n \rightarrow \tau''$

$L \vdash (\ldots, \cdot)\ \texttt{ok}$ Inversion on $\Sigma; A \vdash F, (\ldots, \cdot) : \tau_n \rightarrow \tau''$

$\Sigma; \Gamma \vdash (\ldots, \cdot)\ \texttt{pending}\ \tau_n \rightarrow \texttt{tuple}(\tau_1, \ldots, \tau_n)\ @\ \tau''$
Inversion on $\Sigma; A \vdash F, (\ldots, \cdot) : \tau_n \rightarrow \tau''$

$\Sigma; A \vdash F : \texttt{tuple}(\tau_1, \ldots, \tau_n) \rightarrow \tau''$ Inversion on $\Sigma; A \vdash F, (\ldots, \cdot) : \tau_n \rightarrow \tau''$

$\tau_1\ \texttt{small}\ \ldots \tau_n\ \texttt{small}$
Inversion on $\Sigma; \Gamma \vdash (\ldots, \cdot)\ \texttt{pending}\ \tau_n \rightarrow \texttt{tuple}(\tau_1, \ldots, \tau_n)\ @\ \tau''$

$\ldots = v_i$ where $\Sigma \vdash v_i : \tau_i$
Inversion on $\Sigma; \Gamma \vdash (\ldots, \cdot)\ \texttt{pending}\ \tau_n \rightarrow \texttt{tuple}(\tau_1, \ldots, \tau_n)\ @\ \tau''$

$\texttt{assigned}(F) = A$ Inversion on $\texttt{assigned}(F, (\ldots, \cdot)) = A$

$\texttt{loopnest}(F) = L$ Inversion on $\texttt{loopnest}(F, (\ldots, \cdot)) = L$

$\ldots \Sigma; \Gamma \vdash v_i : \tau_i \ldots$ Rule CHECKEXP-VALUE

$\Sigma; \Gamma \vdash (\ldots, v) : \texttt{tuple}(\tau_1, \ldots, \tau_n)$ Rule CHECKEXP-TUPLE

$\ldots L \vdash v_i\ \texttt{ok}\ \ldots$ Rule CHECKLOOP-VALUE

$L \vdash (\ldots, v)\ \texttt{ok}$ Rule CHECKLOOP-TUPLE

$\triangleleft\ \texttt{tuple}(\ldots, v)\ \texttt{ok}$ Rule DIROK-TUPLE

$\texttt{tuple}(\tau_1, \ldots, \tau_n) = \textbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{tuple}(\ldots, v)\ \texttt{canreturn}\ \tau''$ Vacuously true

$\Sigma \vdash K; F \triangleleft \texttt{tuple}(\ldots, v) : \tau$ Rule CHECKSTATE-NORMAL

$a \notin \texttt{dom}(\mu) \qquad \texttt{allocval}(\mu, a, \tau_a) = \mu'$

$\mu\ \mid K; F \triangleright \texttt{alloc}(\tau_a) \rightarrow \mu'\ \mid K; F \triangleleft \texttt{ptr}(a)$

Note: some trivial steps omitted here.

$\Sigma; \Gamma \vdash \texttt{alloc}(\tau_a) : \tau*$	Inversion on $\Sigma \vdash K; F \rhd \texttt{alloc}(\tau_a) : \tau$
τ_a `alloc`	Inversion on $\Sigma; \Gamma \vdash \texttt{alloc}(\tau_a) : \tau*$
$\exists \mu'.\texttt{allocval}(\mu, a, \tau_a) = \mu'$ and $\mu' : \Sigma'$ and $\Sigma \leq \Sigma'$ and $\Sigma'(a) = \tau_a$.	Lemma 6
$\Sigma' \vdash K; F \rhd \texttt{alloc}(\tau_a) : \tau$	Lemma 10
$\Sigma' \vdash \texttt{ctx}(F) = \Gamma'$	Inversion on $\Sigma' \vdash K; F \rhd \texttt{alloc}(\tau_a) : \tau$
$\Gamma \vdash \texttt{alloc}(\tau_a) : A \to A$	Inversion on $\Sigma' \vdash K; F \rhd \texttt{alloc}(\tau_a) : \tau$
$\Sigma'; A \vdash F : \tau_a* \to \tau''$	Inversion on $\Sigma' \vdash K; F \rhd \texttt{alloc}(\tau_a) : \tau$
$\Sigma' \vdash K : \tau'' \to \tau$	Inversion on $\Sigma' \vdash K; F \rhd \texttt{alloc}(\tau_a) : \tau$
$L \vdash \texttt{ptr}(a)$ `ok`	Rule CHECKLOOP-VALUE
$\Gamma \vdash \texttt{ptr}(a) : A \to A$	Rule CHECKASSIGN-VALUE
$\lhd$ `ptr`(a) `ok`	Rule DIROK-RETURNING
$\Sigma' \vdash \texttt{ptr}(a) : \tau_a*$	Rule CHECKVAL-ADDRESS
$\Sigma'; \Gamma' \vdash \texttt{ptr}(a) : \tau_a*$	Rule CHECKEXP-VALUE
$\tau_a* = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{ptr}(a)$ `canreturn` τ''	Vacuously true
$\Sigma' \vdash K; F \lhd \texttt{ptr}(a) : \tau$	Rule CHECKSTATE-NORMAL

$\mu \mid K; F \rhd \texttt{allocarray}(\tau_a, e) \to \mu' \mid K; F, \texttt{allocarray}(\tau_a, \cdot) \rhd e$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$
$\Sigma; \Gamma \vdash \texttt{allocarray}(\tau_a, e) : \tau_a[]$	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$
$\texttt{assigned}(F) = A$	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$
$\Gamma \vdash \texttt{allocarray}(\tau_a, e) : A \to A$	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$
$\Sigma; A \vdash F : \tau_a[] \to \tau''$	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$
$\texttt{loopnest}(F) = L$	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$
$L \vdash \texttt{allocarray}(\tau_a, e)$ `ok`	Inversion on $\Sigma \vdash K; F \lhd \texttt{allocarray}(\tau_a, e) : \tau$

$\Sigma; \Gamma \vdash e : \mathbf{int}$ and τ_a `alloc`	Inversion on $\Sigma; \Gamma \vdash \texttt{allocarray}(\tau_a, e) : \tau[]$
$\Gamma \vdash \cdot : A \to A$	Rule CHECKASSIGN-HOLE
$\Gamma \vdash \texttt{allocarray}(\tau_a, \cdot) : A \to A$	Rule CHECKASSIGN-ALLOCARRAY
$L \vdash \cdot$ `ok`	Rule CHECKLOOP-VAR
$L \vdash \texttt{allocarray}(\tau_a, \cdot)$ `ok`	Rule CHECKLOOP-ALLOCARRAY
$\Sigma; \Gamma \vdash \texttt{allocarray}(\tau_a, \cdot)$ `pending` $\mathbf{int} \to \mathbf{bool} \ @ \ \tau''$	Rule ISPENDING-ALLOCARRAY
$\Sigma; A \vdash F, \texttt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$	Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, \texttt{allocarray}(\tau_a, \cdot)) = \Gamma$	Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F, \texttt{allocarray}(\tau_a, \cdot)) = A$	Rule GETFRAMEASSIGNED-EXP
$\Gamma \vdash e : A \to A$	Inversion on $\Gamma \vdash \texttt{allocarray}(\tau_a, e) : A \to A$
$L \vdash e$ `ok`	Inversion on $L \vdash \texttt{allocarray}(\tau_a, e)$ `ok`
$\texttt{loopnest}(F, \texttt{allocarray}(\tau_a, \cdot)) = L$	Rule GETLOOPCONTEXT-EXP
$\rhd e$ `ok`	Rule DIROK-PUSHING
$\mathbf{int} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e$ `canreturn` τ''	Vacuously true
$\Sigma \vdash K; F, \texttt{allocarray}(\tau_a, \cdot) \rhd e : \tau$	Rule CHECKSTATE-NORMAL

$$\frac{a \notin \mathtt{dom}(\mu) \qquad \mathtt{makearray}(\mu, a, \tau_a, n) = \mu' \qquad n \geq 0}{\mu \mid K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} \to \mu' \mid K; F \lhd \mathtt{array}(a, n)}$$

$\mathtt{assigned}(F, \mathtt{allocarray}(\tau_a, \cdot)) = A$	
	Inversion on $\Sigma \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$
$\Gamma \vdash \overline{n} : A \to A$	Inversion on $\Sigma \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$
$\Sigma'; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$	
	Inversion on $\Sigma \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$	Inversion on $\Sigma \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$

$\Sigma \vdash \mathtt{ctx}(F) = \Gamma$	Inversion on $\Sigma; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$
$\Sigma; \Gamma \vdash \mathtt{allocarray}(\tau_a, \cdot)\ \mathtt{pending}\ \mathbf{int} \to \tau_a[]\ @\ \tau''$	
	Inversion on $\Sigma; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$
$\tau_a\ \mathtt{alloc}$	Inversion on $\Sigma; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$

$\exists \Sigma'. \mu' : \Sigma'$ and $\Sigma \leq \Sigma'$ and $\Sigma' \vdash \mathtt{array}(a, n) : \tau_a[]$	Lemma 9
$\Sigma' \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$	Lemma 10
$\mathtt{assigned}(F, \mathtt{allocarray}(\tau_a, \cdot)) = A$	
	Inversion on $\Sigma' \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$
$\Gamma \vdash \overline{n} : A \to A$	Inversion on $\Sigma' \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$
$\Sigma'; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau_a[]$	
	Inversion on $\Sigma' \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$
$\Sigma' \vdash K : \tau'' \to \tau$	Inversion on $\Sigma' \vdash K; F, \mathtt{allocarray}(\tau_a, \cdot) \lhd \overline{n} : \tau$

$\Sigma' \vdash \mathtt{ctx}(F) = \Gamma$	Inversion on $\Sigma'; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$
$\Gamma \vdash \mathtt{allocarray}(\tau_a, \cdot) : A \to A$	Inversion on $\Sigma'; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$
$\Sigma'; \Gamma \vdash \mathtt{allocarray}(\tau_a, \cdot)\ \mathtt{pending}\ \mathbf{int} \to \tau_a[]\ @\ \tau''$	
	Inversion on $\Sigma'; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$
$\mathtt{loopnest}(F) = L$	Inversion on $\Sigma'; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$
$\Sigma'; \Gamma \vdash F : \tau_a[] \to \tau''$	Inversion on $\Sigma'; A \vdash F, \mathtt{allocarray}(\tau_a, \cdot) : \mathbf{int} \to \tau''$

$\mathtt{assigned}(F) = A$	Inversion on $\mathtt{assigned}(F, \mathtt{allocarray}(\tau_a, \cdot)) = A$
$L \vdash \mathtt{array}(a, n)\ \mathtt{ok}$	Rule CHECKLOOP-VALUE
$\Gamma \vdash \mathtt{array}(a, n) : A \to A$	Rule CHECKASSIGN-VALUE
$\lhd \mathtt{array}(a, n)\ \mathtt{ok}$	Rule DIROK-RETURNING
$\tau_a[] = \mathbf{cmd} \Rightarrow \Sigma'; \Gamma \vdash \mathtt{array}(a, n)\ \mathtt{canreturn}\ \tau''$	Vacuously true
$\Sigma' \vdash K; F \lhd \mathtt{array}(a, n) : \tau$	Rule CHECKSTATE-NORMAL

$$\frac{n < 0}{\mu \mid K; F, \mathtt{allocarray}(\tau, \cdot) \lhd \overline{n} \to \mu \mid K; F \lhd \mathtt{exn}}$$

$\Sigma \vdash K; F \lhd \texttt{exn} : \tau$ Rule CHECKSTATE-EXN

$\mu \mid K; F \rhd \texttt{if}(e_c, e_t, e_f) \to \mu \mid K; F, \texttt{if}(\cdot, e_t, e_f) \rhd e_c$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f) : \tau'$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\texttt{assigned}(F) = A$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\Gamma \vdash \texttt{if}(e_c, e_t, e_f) : A \to A'$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\Sigma; \Gamma \vdash e_c : \mathbf{bool}$ and $\Sigma; \Gamma \vdash e_t : \tau'$ and $\Sigma; \Gamma \vdash e_f : \tau'$
Inversion on $\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f) : \tau'$

$\Gamma \vdash e_c : A \to A$ and $\Gamma \vdash e_t : A \to A_t$ and $\Gamma \vdash e_f : A \to A_f$ where $A_t \cap A_f = A'$
Inversion on $\Gamma \vdash \texttt{if}(e_c, e_t, e_f) : A \to A'$

$\Gamma \vdash \cdot : A \to A$ Rule CHECKASSIGN-HOLE

$\Gamma \vdash \texttt{if}(\cdot, e_t, e_f) : A \to A'$ Rule CHECKASSIGN-IF

$\texttt{loopnest}(F) = L$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$L \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

There are two possible rules used for the derivation of $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$
Case CHECKSTATE-NORMAL

$\Sigma; A' \vdash F : \tau' \to \tau''$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{canreturn}\ \tau''$
Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$L \vdash e_t\ \texttt{ok}$ and $L \vdash e_f\ \texttt{ok}$ and $L \vdash e_c\ \texttt{ok}$ Inversion on $L \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{ok}$

$L \vdash \cdot\ \texttt{ok}$ Rule CHECKLOOP-VAR

$L \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{ok}$ Rule CHECKLOOP-IF

$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_t\ \texttt{canreturn}\ \tau''$ and $\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_f\ \texttt{canreturn}\ \tau''$
Inversion on $\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{canreturn}\ \tau''$

$\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{pending}\ \mathbf{bool} \to \tau' \mathbin{@} \tau''$ Rule ISPENDING-IF

$\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET

Otherwise CHECKSTATE-RETURNS was used:

$\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{returns}\ \tau''$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$L = \texttt{notinloop}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\tau' = \mathbf{cmd}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{if}(e_c, e_t, e_f) : \tau$

$\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{canreturn}\ \tau''$
Inversion on $\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{returns}\ \tau''$

$\Sigma; \Gamma \vdash e_t\ \texttt{canreturn}\ \tau''$ and $\Sigma; \Gamma \vdash e_f\ \texttt{canreturn}\ \tau''$
Inversion on $\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{canreturn}\ \tau''$

$\vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{returns}$ Inversion on $\Sigma; \Gamma \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{returns}\ \tau''$

$\vdash e_t\ \texttt{returns}$ and $\vdash e_f\ \texttt{returns}$ Inversion on $\vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{returns}$

$\vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{returns}$ Rule DOESRETURN-IF

$\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{pending}\ \mathbf{bool} \to \mathbf{cmd} \mathbin{@} \tau''$ Rule ISPENDING-IF
$\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPRET
$\Sigma \vdash \texttt{ctx}(F, \texttt{if}(\cdot, e_t, e_f)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F, \texttt{if}(\cdot, e_t, e_f)) = A$ Rule GETFRAMEASSIGNED-EXP
$\texttt{loopnest}(F, \texttt{if}(\cdot, e_t, e_f)) = L$ Rule GETLOOPCONTEXT-EXP
$L \vdash e_c\ \texttt{ok}$ Inversion on $L \vdash \texttt{if}(e_c, e_t, e_f)\ \texttt{ok}$
$\rhd e_c\ \texttt{ok}$ Rule DIROK-PUSHING
$\mathbf{bool} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_c\ \texttt{canreturn}\ \tau''$ Vacuously true
$\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \rhd e_c : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F, \texttt{if}(\cdot, e_t, e_f) \lhd \mathbf{true} \to \mu \mid K; F \rhd e_t$

$\Sigma \vdash \texttt{ctx}(F, \texttt{if}(\cdot, e_t, e_f)) = \Gamma$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Sigma; \Gamma \vdash \mathbf{true} : \mathbf{bool}$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\texttt{assigned}(F, \texttt{if}(\cdot, e_t, e_f)) = A$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Gamma \vdash \mathbf{true} : A \to A$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash \texttt{ctx}(F, \texttt{if}(\cdot, e_t, e_f)) = \Gamma$
$\texttt{assigned}(F) = A$ Inversion on $\texttt{assigned}(F, \texttt{if}(\cdot, e_t, e_f)) = A$
$\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{pending}\ \mathbf{bool} \to \tau' \mathbin{@} \tau''$
Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Sigma; \Gamma \vdash e_t : \tau'$ Inversion on $\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{pending}\ \mathbf{bool} \to \tau' \mathbin{@} \tau''$
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_t\ \texttt{canreturn}\ \tau''$
Inversion on $\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{pending}\ \mathbf{bool} \to \tau' \mathbin{@} \tau''$
$\Gamma \vdash \texttt{if}(\cdot, e_t, e_f) : A \to A'$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Gamma \vdash e_t : A \to A_t$ and $\Gamma \vdash e_f : A \to A_f$ where $A_t \cap A_f = A'$
Inversion on $\Gamma \vdash \texttt{if}(\cdot, e_t, e_f) : A \to A'$
There are two possible rules used for the derivation of $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
Case CHECKFRAMETYPE-FRAMEEXPNORET
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$L \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{ok}$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Sigma; A' \vdash F : \tau' \to \tau''$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Sigma; A_t \vdash F : \tau' \to \tau''$ Lemma 18
$L \vdash e_t\ \texttt{ok}$ Inversion on $L \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{ok}$
$\rhd e_t\ \texttt{ok}$ Rule DIROK-PUSHING
$\Sigma \vdash K; F \rhd e_t : \tau$ Rule CHECKSTATE-NORMAL
Otherwise CHECKFRAMETYPE-FRAMEEXPRET
$\texttt{loopnest}(F) = \texttt{notinloop}$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\texttt{notinloop} \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{ok}$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{returns}$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$

$\tau' = \mathbf{cmd}$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\vdash e_t$ `returns` Inversion on $\vdash \texttt{if}(\cdot, e_t, e_f)$ `returns`
$\Sigma; \Gamma \vdash e_t$ `returns` τ'' Rule RETURNS-ONLY
`notinloop` $\vdash e_t$ `ok` Inversion on `notinloop` $\vdash \texttt{if}(\cdot, e_t, e_f)$ `ok`
$\rhd e_t$ `ok` Rule DIROK-PUSHING
$\Sigma \vdash K; F \rhd e_t : \tau$ Rule CHECKSTATE-RETURNS

$\mu \mid K; F, \texttt{if}(\cdot, e_t, e_f) \lhd \mathbf{false} \to \mu \mid K; F \rhd e_f$

$\Sigma \vdash \texttt{ctx}(F, \texttt{if}(\cdot, e_t, e_f)) = \Gamma$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Sigma; \Gamma \vdash \mathbf{false} : \mathbf{bool}$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\texttt{assigned}(F, \texttt{if}(\cdot, e_t, e_f)) = A$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Gamma \vdash \mathbf{false} : A \to A$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, \texttt{if}(\cdot, e_t, e_f) \lhd v : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash \texttt{ctx}(F, \texttt{if}(\cdot, e_t, e_f)) = \Gamma$
$\texttt{assigned}(F) = A$ Inversion on $\texttt{assigned}(F, \texttt{if}(\cdot, e_t, e_f)) = A$
$\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)$ `pending` $\mathbf{bool} \to \tau' \mathbin{@} \tau''$
Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Sigma; \Gamma \vdash e_f : \tau'$ Inversion on $\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)$ `pending` $\mathbf{bool} \to \tau' \mathbin{@} \tau''$
$\tau' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e_f$ `canreturn` τ''
Inversion on $\Sigma; \Gamma \vdash \texttt{if}(\cdot, e_t, e_f)$ `pending` $\mathbf{bool} \to \tau' \mathbin{@} \tau''$
$\Gamma \vdash \texttt{if}(\cdot, e_t, e_f) : A \to A'$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Gamma \vdash e_t : A \to A_t$ and $\Gamma \vdash e_f : A \to A_f$ where $A_t \cap A_f = A'$
Inversion on $\Gamma \vdash \texttt{if}(\cdot, e_t, e_f) : A \to A'$
There are two possible rules used for the derivation of $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
Case CHECKFRAMETYPE-FRAMEEXPNORET
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$L \vdash \texttt{if}(\cdot, e_t, e_f)$ `ok` Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Sigma; A' \vdash F : \tau' \to \tau''$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\Sigma; A_f \vdash F : \tau' \to \tau''$ Lemma 18
$L \vdash e_f$ `ok` Inversion on $L \vdash \texttt{if}(\cdot, e_t, e_f)$ `ok`
$\rhd e_f$ `ok` Rule DIROK-PUSHING
$\Sigma \vdash K; F \rhd e_f : \tau$ Rule CHECKSTATE-NORMAL
Otherwise CHECKFRAMETYPE-FRAMEEXPRET
$\texttt{loopnest}(F) =$ `notinloop` Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
`notinloop` $\vdash \texttt{if}(\cdot, e_t, e_f)$ `ok` Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\vdash \texttt{if}(\cdot, e_t, e_f)$ `returns` Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\tau' = \mathbf{cmd}$ Inversion on $\Sigma; A \vdash F, \texttt{if}(\cdot, e_t, e_f) : \mathbf{bool} \to \tau''$
$\vdash e_f$ `returns` Inversion on $\vdash \texttt{if}(\cdot, e_t, e_f)$ `returns`
$\Sigma; \Gamma \vdash e_f$ `returns` τ'' Rule RETURNS-ONLY

$\texttt{notinloop} \vdash e_f\ \texttt{ok}$ Inversion on $\texttt{notinloop} \vdash \texttt{if}(\cdot, e_t, e_f)\ \texttt{ok}$
$\rhd e_f\ \texttt{ok}$ Rule DIROK-PUSHING
$\Sigma \vdash K; F \rhd e_f : \tau$ Rule CHECKSTATE-RETURNS

$\mu \mid K; F \rhd \texttt{decl}(x, \tau, e) \to \mu \mid K; F, x : \tau \rhd e$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\Sigma; \Gamma \vdash \texttt{decl}(x, \tau_x, e) : \mathbf{cmd}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\texttt{assigned}(F) = A$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\Gamma \vdash \texttt{decl}(x, \tau_x, e) : A \to A' - \{x\}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$x \notin A$ and $\Gamma, x : \tau_x \vdash e : A \to A'$ Inversion on $\Gamma \vdash \texttt{decl}(x, \tau, e) : A \to A' - \{x\}$
$\Sigma; \Gamma, x : \tau_x \vdash e : \mathbf{cmd}$ Inversion on $\Sigma; \Gamma \vdash \texttt{decl}(x, \tau_x, e) : \mathbf{cmd}$
There are two possible rules used for the derivation of $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
Case CHECKSTATE-NORMAL:
$\Sigma; A' - \{x\} \vdash F : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$L \vdash \texttt{decl}(x, \tau_x, e)\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{decl}(x, \tau, e)\ \texttt{canreturn}\ \tau''$
Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\Sigma; A' \vdash F, x : \tau_x : \mathbf{cmd} \to \tau''$ Rule CHECKFRAMETYPE-FRAMEVAR
$\Sigma \vdash \texttt{ctx}(F, x : \tau_x) = \Gamma, x : \tau_x$ Rule GETFRAMECONTEXT-VAR
$\texttt{assigned}(F, x : \tau_x) = A$ Rule GETFRAMEASSIGNED-VAR
$\texttt{loopnest}(F, x : \tau_x) = L$ Rule GETLOOPCONTEXT-DECL
$L \vdash e\ \texttt{ok}$ Inversion on $L \vdash \texttt{decl}(x, \tau_x, e)\ \texttt{ok}$
$\rhd e\ \texttt{ok}$ Rule DIROK-PUSHING
$\Sigma; \Gamma, x : \tau_x \vdash e\ \texttt{canreturn}\ \tau''$ Inversion on $\Sigma; \Gamma \vdash \texttt{decl}(x, \tau_x, e)\ \texttt{canreturn}\ \tau''$
$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma, x : \tau_x \vdash e\ \texttt{canreturn}\ \tau''$ Weakening
$\Sigma \vdash K; F, x : \tau_x \rhd e : \tau$ Rule CHECKSTATE-NORMAL
Otherwise CHECKSTATE-RETURNS was used:
$\texttt{loopnest}(F) = \texttt{notinloop}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\texttt{notinloop} \vdash \texttt{decl}(x, \tau_x, e)\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\Sigma; \Gamma \vdash \texttt{decl}(x, \tau, e)\ \texttt{returns}\ \tau''$ Inversion on $\Sigma \vdash K; F \rhd \texttt{decl}(x, \tau_x, e) : \tau$
$\vdash \texttt{decl}(x, \tau_x, e)\ \texttt{returns}$ Inversion on $\Sigma; \Gamma \vdash \texttt{decl}(x, \tau, e)\ \texttt{returns}\ \tau''$
$\vdash e\ \texttt{returns}$ Inversion on $\vdash \texttt{decl}(x, \tau_x, e)\ \texttt{returns}$
$\tau''\ \texttt{small}$ Inversion on $\Sigma; \Gamma \vdash \texttt{decl}(x, \tau, e)\ \texttt{returns}\ \tau''$
$\Sigma; \Gamma \vdash \texttt{decl}(x, \tau_x, e)\ \texttt{canreturn}\ \tau''$
Inversion on $\Sigma; \Gamma \vdash \texttt{decl}(x, \tau, e)\ \texttt{returns}\ \tau''$
$\Sigma; \Gamma, x : \tau_x \vdash e\ \texttt{canreturn}\ \tau''$ Inversion on $\Sigma; \Gamma \vdash \texttt{decl}(x, \tau_x, e)\ \texttt{canreturn}\ \tau''$
$\Sigma; \Gamma, x : \tau_x \vdash e\ \texttt{returns}\ \tau''$ Rule RETURNS-ONLY
$\texttt{loopnest}(F, x : \tau_x) = \texttt{notinloop}$ Rule GETLOOPCONTEXT-DECL
$\texttt{notinloop} \vdash e\ \texttt{ok}$ Inversion on $\texttt{notinloop} \vdash \texttt{decl}(x, \tau_x, e)\ \texttt{ok}$

$\triangleright\ e\ \texttt{ok}$ Rule DIROK-PUSHING
$\Sigma \vdash \texttt{ctx}(F, x : \tau_x) = \Gamma, x : \tau_x$ Rule GETFRAMECONTEXT-VAR
$\texttt{assigned}(F, x : \tau_x) = A$ Rule GETFRAMEASSIGNED-VAR
$\Sigma \vdash K; F, x : \tau_x \triangleright e : \tau$ Rule CHECKSTATE-RETURNS

$\mu \mid K; F, x : \tau \triangleleft \texttt{nop} \to \mu \mid K; F \triangleleft \texttt{nop}$

$\texttt{assigned}(F, x : \tau_x) = A$ Inversion on $\Sigma \vdash K; F, x : \tau_x \triangleleft \texttt{nop} : \tau$
$\Gamma \vdash \texttt{nop} : A \to A$ Inversion on $\Sigma \vdash K; F, x : \tau_x \triangleleft \texttt{nop} : \tau$
$\Sigma; A \vdash F, x : \tau_x : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma \vdash K; F, x : \tau_x \triangleleft \texttt{nop} : \tau$
$\texttt{loopnest}(F, x : \tau_x) = L$ Inversion on $\Sigma \vdash K; F, x : \tau_x \triangleleft \texttt{nop} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F, x : \tau_x \triangleleft \texttt{nop} : \tau$
$\texttt{assigned}(F) = A$ and $x \notin A$ Inversion on $\texttt{assigned}(F, x : \tau_x) = A$
$\Sigma; A - \{x\} \vdash F : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma; A \vdash F, x : \tau_x : \mathbf{cmd} \to \tau''$
$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ $x \notin A \Rightarrow A - \{x\} = A$
$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ for some Γ By construction
$\Sigma \vdash \texttt{nop} : \mathbf{cmd}$ Rule CHECKVAL-NOP
$\Sigma; \Gamma \vdash \texttt{nop} : \mathbf{cmd}$ Rule CHECKEXP-VALUE
$\texttt{loopnest}(F) = L$ Inversion on $\texttt{loopnest}(F, x : \tau_x) = L$
$\Sigma; \Gamma \vdash \texttt{nop}\ \texttt{canreturn}\ \tau''$ Rule ONLYRETURNS-NOP
$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{nop}\ \texttt{canreturn}\ \tau''$ Weakening
$\Sigma \vdash K; F \triangleleft \texttt{nop} : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F \triangleright \texttt{assign}(x, e) \to \mu \mid K; F, \texttt{assign}(x, \cdot) \triangleright e$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$\Sigma; \Gamma \vdash \texttt{assign}(x, e) : \mathbf{cmd}$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$\texttt{assigned}(F) = A$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$\Gamma \vdash \texttt{assign}(x, e) : A \to A \cup \{x\}$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$L \vdash \texttt{assign}(x, e)\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$\Sigma; A \cup \{x\} \vdash F : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F \triangleright \texttt{assign}(x, e) : \tau$
$\Gamma = \Gamma', x : \tau'$ and $\Sigma; \Gamma \vdash e : \tau'$ and $\tau'\ \texttt{small}$ Inversion on $\Sigma; \Gamma \vdash \texttt{assign}(x, e) : \mathbf{cmd}$
$\Gamma \vdash e : A \to A$ Inversion on $\Gamma \vdash \texttt{assign}(x, e) : A \to A \cup \{x\}$
$L \vdash e\ \texttt{ok}$ Inversion on $L \vdash \texttt{assign}(x, e)\ \texttt{ok}$
$\Gamma \vdash \cdot : A \to A$ Rule CHECKASSIGN-HOLE
$\Gamma \vdash \texttt{assign}(x, \cdot) : A \to A \cup \{x\}$ Rule CHECKASSIGN-ASSIGN
$L \vdash \cdot\ \texttt{ok}$ Rule CHECKLOOP-VAR
$L \vdash \texttt{assign}(x, \cdot)\ \texttt{ok}$ Rule CHECKLOOP-ASSIGN
$\Sigma; \Gamma \vdash x : \tau'$ Rule CHECKEXP-VAR

$\Sigma; \Gamma \vdash \texttt{assign}(x, \cdot)\ \texttt{pending}\ \tau' \to \mathbf{cmd}\ @\ \tau''$ Rule ISPENDING-ASSIGN

$\Sigma; A \vdash F, \texttt{assign}(x, \cdot) : \tau' \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, \texttt{assign}(x, \cdot)) = \Gamma$ Rule GETFRAMECONTEXT-EXP

$\texttt{assigned}(F, \texttt{assign}(x, \cdot)) = A$ Rule GETFRAMEASSIGNED-EXP

$\texttt{loopnest}(F, \texttt{assign}(x, \cdot)) = L$ Rule GETLOOPCONTEXT-EXP

$\triangleright\ e\ \texttt{ok}$ Rule DIROK-PUSHING

Since τ'' `small` by inversion $\tau'' \neq \mathbf{cmd}$.

$\tau'' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e\ \texttt{canreturn}\ \tau''$ Vacuously true

$\Sigma \vdash K; F, \texttt{assign}(x, \cdot) \triangleright e : \tau$ Rule CHECKSTATE-NORMAL

$\mu\ \mid K; F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' \to \mu\ \mid K; F, x : \tau_x = v', \ldots \triangleleft \texttt{nop}$

$\Sigma \vdash \texttt{ctx}(F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot)) = \Gamma$

$\Sigma \vdash K; F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$ Inversion on $\Sigma \vdash K; F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$

$\Sigma; \Gamma \vdash v' : \tau'$ Inversion on $\Sigma \vdash K; F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$

$\texttt{assigned}(F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot)) = A$

$\Sigma \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$ Inversion on $\Sigma \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$

$\Gamma \vdash v' : A \to A$ Inversion on $\Sigma \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$

$\Sigma; A \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) : \tau' \to \tau''$

$\Sigma \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$ Inversion on $\Sigma \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$

$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' : \tau$

$\Sigma; \Gamma \vdash \texttt{assign}(x, \cdot)\ \texttt{pending}\ \tau' \to \mathbf{cmd}\ @\ \tau''$

$\Sigma; A \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) : \tau' \to \tau''$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) : \tau' \to \tau''$

$\Sigma; \Gamma \vdash x : \tau'$ Inversion on $\Sigma; \Gamma \vdash \texttt{assign}(x, \cdot)\ \texttt{pending}\ \tau' \to \mathbf{cmd}\ @\ \tau''$

$\Sigma; \Gamma', x : \tau' \vdash x : \tau'$ where $\Gamma = \Gamma', x : \tau'$ Inversion on $\Sigma; \Gamma \vdash x : \tau'$

$\Sigma \vdash v' : \tau'$ Inversion on $\Sigma; \Gamma \vdash v' : \tau'$

$\Sigma \vdash v : \tau'$ and $\tau' = \tau_x$ Lemma 5

$\texttt{assigned}(F, x : \tau_x = v', \ldots) = A$ Lemma 20

$\Sigma; A \vdash F, x : \tau_x = v, \ldots : \mathbf{cmd} \to \tau''$

$\Sigma; A \vdash F, x : \tau_x = v', \ldots : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) : \tau' \to \tau''$

$\Sigma \vdash \texttt{ctx}(F, x : \tau_x = v', \ldots) = \Gamma''$ for some Γ'' Lemma 24

$\Sigma; \Gamma'' \vdash \texttt{nop} : \mathbf{cmd}$ By construction

$\Gamma \vdash \texttt{nop} : A \to A$ Rule CHECKEXP-VALUE

$\texttt{loopnest}(F, x : \tau_x = v', \ldots) = L$ for some L Rule CHECKASSIGN-VAR

$L \vdash \texttt{nop}\ \texttt{ok}$ By construction

$\triangleleft\ \texttt{nop}\ \texttt{ok}$ Rule CHECKLOOP-VALUE

$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{nop}\ \texttt{canreturn}\ \tau''$ Rule DIROK-RETURNING

$\Sigma \vdash K; F, x : \tau_x = v', \ldots \triangleleft \texttt{nop} : \tau$ Weakening of Rule ONLYRETURNS-NOP

Rule CHECKSTATE-NORMAL

$\mu\ \mid K; F, x : \tau_x, \ldots, \texttt{assign}(x, \cdot) \triangleleft v' \to \mu\ \mid K; F, x : \tau_x = v', \ldots \triangleleft \texttt{nop}$

$\Sigma \vdash \texttt{ctx}(F, x : \tau, \ldots, \texttt{assign}(x, \cdot)) = \Gamma$

Inversion on $\Sigma \vdash K; F, x : \tau, \ldots, \texttt{assign}(x, \cdot) \lhd v' : \tau$

$\Sigma; \Gamma \vdash v' : \tau'$ Inversion on $\Sigma \vdash K; F, x : \tau, \ldots, \texttt{assign}(x, \cdot) \lhd v' : \tau$

$\texttt{assigned}(F, x : \tau, \ldots, \texttt{assign}(x, \cdot)) = A$

Inversion on $\Sigma \vdash F, x : \tau, \ldots, \texttt{assign}(x, \cdot) \lhd v' : \tau$

$\Gamma \vdash v' : A \to A$ Inversion on $\Sigma \vdash F, x : \tau, \ldots, \texttt{assign}(x, \cdot) \lhd v' : \tau$

$\Sigma; A \vdash F, x : \tau, \ldots, \texttt{assign}(x, \cdot) : \tau' \to \tau''$

Inversion on $\Sigma \vdash F, x : \tau, \ldots, \texttt{assign}(x, \cdot) \lhd v' : \tau$

$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash F, x : \tau_x = v, \ldots, \texttt{assign}(x, \cdot) \lhd v' : \tau$

$\Sigma; \Gamma \vdash \texttt{assign}(x, \cdot)\ \texttt{pending}\ \tau' \to \mathbf{cmd} \mathbin{@} \tau''$

Inversion on $\Sigma; A \vdash F, x : \tau, \ldots, \texttt{assign}(x, \cdot) : \tau' \to \tau''$

$\Sigma; \Gamma \vdash x : \tau'$ Inversion on $\Sigma; \Gamma \vdash \texttt{assign}(x, \cdot)\ \texttt{pending}\ \tau' \to \mathbf{cmd} \mathbin{@} \tau''$

$\Sigma; \Gamma', x : \tau' \vdash x : \tau'$ where $\Gamma = \Gamma', x : \tau'$ Inversion on $\Sigma; \Gamma \vdash x : \tau'$

$\Sigma \vdash v' : \tau'$ Inversion on $\Sigma; \Gamma \vdash v' : \tau'$

$\Sigma \vdash v' : \tau_x$ Lemma 5

$\texttt{assigned}(F, x : \tau = v', \ldots) = A \cup \{x\}$ Lemma 19

$\Sigma; A \cup \{x\} \vdash F, x : \tau, \ldots : \mathbf{cmd} \to \tau''$

Inversion on $\Sigma; A \vdash F, x : \tau, \ldots, \texttt{assign}(x, \cdot) : \tau' \to \tau''$

$\Sigma; A \cup \{x\} \vdash F, x : \tau = v', \ldots : \mathbf{cmd} \to \tau''$ Lemma 23

$\Sigma \vdash \texttt{ctx}(F, x : \tau = v', \ldots) = \Gamma''$ for some Γ'' By construction

$\Sigma; \Gamma'' \vdash \texttt{nop} : \mathbf{cmd}$ Rule CHECKEXP-VALUE

$\Gamma \vdash \texttt{nop} : A \cup \{x\} \to A \cup \{x\}$ Rule CHECKASSIGN-VAR

$\texttt{loopnest}(F, x : \tau = v', \ldots) = L$ for some L By construction

$L \vdash \texttt{nop ok}$ Rule CHECKLOOP-VALUE

$\lhd \texttt{nop ok}$ Rule DIROK-RETURNING

$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{nop canreturn}\ \tau''$ Weakening of Rule ONLYRETURNS-NOP

$\Sigma \vdash K; F, x : \tau = v', \ldots \lhd \texttt{nop} : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F, x : \tau_x = v \lhd \texttt{nop} \to \mu \mid K; F \lhd \texttt{nop}$

$\texttt{assigned}(F, x : \tau_x = v) = A$ Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{nop} : \tau$

$\Gamma \vdash \texttt{nop} : A \to A$ Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{nop} : \tau$

$\Sigma; A \vdash F, x : \tau_x = v : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{nop} : \tau$

$\Sigma; A - \{x\} \vdash F : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v : \mathbf{cmd} \to \tau''$

$\texttt{assigned}(F) = A'$ where $A' \cup \{x\} = A$ and $x \notin A'$

Inversion on $\texttt{assigned}(F, x : \tau_x = v) = A$

$A - \{x\} = A'$ $A' \cup \{x\} = A$ and $x \notin A$

$\Gamma \vdash \texttt{nop} : A' \to A'$ Rule CHECKASSIGN-VALUE

$\Sigma \vdash \texttt{ctx}(F, x : \tau_x = v) = \Gamma$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v : \mathbf{cmd} \to \tau''$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash \texttt{ctx}(F, x : \tau_x = v) = \Gamma$

$\Sigma \vdash \texttt{nop} : \mathbf{cmd}$ Rule CHECKVAL-NOP

$\Sigma; \Gamma \vdash \texttt{nop} : \mathbf{cmd}$ Rule CHECKEXP-VALUE

$\texttt{loopnest}(F, x : \tau_x = v) = L$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v : \mathbf{cmd} \to \tau''$

$\texttt{loopnest}(F) = L$ Inversion on $\texttt{loopnest}(F, x : \tau_x = v) = L$
$L \vdash \texttt{nop ok}$ Rule CHECKLOOP-VALUE
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v : \mathbf{cmd} \to \tau''$
$\lhd \texttt{nop ok}$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v : \mathbf{cmd} \to \tau''$
$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{nop canreturn } \tau''$ Inversion on $\Sigma; A \vdash F, x : \tau_x = v : \mathbf{cmd} \to \tau''$
$\Sigma \vdash K; F \lhd \texttt{nop} : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F \rhd \texttt{return}(e) \to \mu \mid K; F, \texttt{return}(\cdot) \rhd e$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\Sigma; \Gamma \vdash \texttt{return}(e) : \mathbf{cmd}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\texttt{assigned}(F) = A$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{return}(e) : A \to \{x_1, \ldots, x_n\}$
Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
There are two possible rules used for the derivation of $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
Case CHECKSTATE-NORMAL:
$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{return}(e) \texttt{ canreturn } \tau''$
Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\Sigma; \Gamma \vdash \texttt{return}(e) \texttt{ canreturn } \tau''$ Modus ponens
$\Sigma; \Gamma \vdash e : \tau''$ and $\tau'' \texttt{ small}$ Inversion on $\Sigma; \Gamma \vdash \texttt{return}(e) \texttt{ canreturn } \tau''$
$\texttt{loopnest}(F) = L$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$L \vdash \texttt{return}(e) \texttt{ ok}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\vdash \texttt{return}(\cdot) \texttt{ returns}$ Rule DOESRETURN-RETURN
$\Gamma \vdash \cdot : A \to A$ Rule CHECKASSIGN-HOLE
$\Gamma \vdash \texttt{return}(\cdot) : A \to A$ Rule CHECKASSIGN-RETURN
$L \vdash \texttt{return}(\cdot) \texttt{ ok}$ Rule CHECKLOOP-RETURN
$\Sigma; \Gamma \vdash \texttt{return}(\cdot) \texttt{ pending } \tau'' \to \mathbf{cmd} \mathbin{@} \tau''$ Rule ISPENDING-RETURN
$\Sigma; A \vdash F, \texttt{return}(\cdot) : \tau'' \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPNORET
$\Sigma \vdash \texttt{ctx}(F, \texttt{return}(\cdot)) = \Gamma$ Rule GETFRAMECONTEXT-EXP
$\texttt{assigned}(F, \texttt{return}(\cdot)) = A$ Rule GETFRAMEASSIGNED-EXP
$\Gamma \vdash e : A \to A$ Inversion on $\Gamma \vdash \texttt{return}(e) : A \to A$
$\texttt{loopnest}(F, \texttt{return}(\cdot)) = L$ Rule GETLOOPCONTEXT-EXP
$L \vdash e \texttt{ ok}$ Inversion on $L \vdash \texttt{return}(e) \texttt{ ok}$
$\rhd e \texttt{ ok}$ Rule DIROK-PUSHING
Since $\tau'' \texttt{ small}$ by inversion $\tau'' \neq \mathbf{cmd}$.
$\tau'' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e \texttt{ canreturn } \tau''$ Vacuously true
$\Sigma \vdash K; F, \texttt{return}(\cdot) \rhd e : \tau$ Rule CHECKSTATE-NORMAL
Otherwise CHECKSTATE-RETURNS was used
$\texttt{loopnest}(F) = \texttt{ notinloop}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\texttt{notinloop} \vdash \texttt{return}(e) \texttt{ ok}$ Inversion on $\Sigma \vdash K; F \rhd \texttt{return}(e) : \tau$
$\Sigma; \Gamma \vdash e : \tau''$ Inversion on $\Sigma; \Gamma \vdash e : \tau''$

$\vdash \texttt{return}(\cdot)\ \texttt{returns}$ Rule DOESRETURN-RETURN

$\Gamma \vdash \cdot : A \to A$ Rule CHECKASSIGN-HOLE

$\Gamma \vdash \texttt{return}(\cdot) : A \to A$ Rule CHECKASSIGN-RETURN

$\texttt{notinloop} \vdash \texttt{return}(\cdot)\ \texttt{ok}$ Rule CHECKLOOP-RETURN

$\Sigma; \Gamma \vdash \texttt{return}(\cdot)\ \texttt{pending}\ \tau'' \to \mathbf{cmd} \mathbin{@} \tau''$ Rule ISPENDING-RETURN

$\Sigma; A \vdash F, \texttt{return}(\cdot) : \tau'' \to \tau''$ Rule CHECKFRAMETYPE-FRAMEEXPRET

$\Sigma \vdash \texttt{ctx}(F, \texttt{return}(\cdot)) = \Gamma$ Rule GETFRAMECONTEXT-EXP

$\texttt{assigned}(F, \texttt{return}(\cdot)) = A$ Rule GETFRAMEASSIGNED-EXP

$\Gamma \vdash e : A \to A$ Inversion on $\Gamma \vdash \texttt{return}(e) : A \to A$

$\texttt{loopnest}(F, \texttt{return}(\cdot)) = \texttt{notinloop}$ Rule GETLOOPCONTEXT-EXP

$\texttt{notinloop} \vdash e\ \texttt{ok}$ Inversion on $\texttt{notinloop} \vdash \texttt{return}(e)\ \texttt{ok}$

$\rhd e\ \texttt{ok}$ Rule DIROK-PUSHING

Since $\tau''\ \texttt{small}$ by inversion $\tau'' \neq \mathbf{cmd}$.

$\tau'' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e\ \texttt{canreturn}\ \tau''$ Vacuously true

$\Sigma \vdash K; F, \texttt{return}(\cdot) \rhd e : \tau$ Rule CHECKSTATE-NORMAL

$\mu \mid K; F, \texttt{return}(\cdot) \lhd e \to \mu \mid K \lhd e$

$\Sigma; \Gamma \vdash e : \tau'$ Inversion on $\Sigma \vdash K; F, \texttt{return}(\cdot) \lhd e : \tau$

$\Sigma; A \vdash K; F, \texttt{return}(\cdot) : \tau' \to \tau''$ Inversion on $\Sigma \vdash K; F, \texttt{return}(\cdot) \lhd e : \tau$

$\lhd e\ \texttt{ok}$ Inversion on $\Sigma \vdash K; F, \texttt{return}(\cdot) \lhd e : \tau$

$\Sigma; \Gamma \vdash \texttt{return}(\cdot)\ \texttt{pending}\ \tau' \to \mathbf{cmd} \mathbin{@} \tau''$
Inversion on $\Sigma; A \vdash K; F, \texttt{return}(\cdot) : \tau' \to \tau''$

$\tau' = \tau''$ and $\tau''\ \texttt{small}$ Inversion on $\Sigma; \Gamma \vdash \texttt{return}(\cdot)\ \texttt{pending}\ \tau' \to \mathbf{cmd} \mathbin{@} \tau''$

$\Sigma \vdash K : \tau'' \to \tau$ Inversion on $\Sigma; \Gamma \vdash e : \tau'$

Case analysis on the derivation of $\Sigma \vdash K : \tau'' \to \tau$ gives two possible cases.

If the rule `checkstack-empty` was used then

$K = \cdot$ and $\tau = \tau''$ Inversion on $\Sigma \vdash K : \tau'' \to \tau$

Via case analysis on $\lhd e\ \texttt{ok}$: either $e =$ some v or $e = ()$.

If $e = v$ then

$\Sigma; \cdot \vdash v : \tau$ Rule CHECKEXP-VALUE

Otherwise $e = ()$

$\tau' = ()$ Inversion on $\Sigma; \Gamma \vdash e : \tau'$

$\Sigma; \cdot \vdash () : \tau$ Rule CHECKEXP-TUPLE

$\Sigma \vdash \cdot \lhd e : \tau$ Rule CHECKSTATE-EMPTY

Otherwise the rule `checkstack-nonempty` was used.

$K = K'; F'$ and $\texttt{assigned}(F') = A'$ and $\Sigma; A' \vdash F' : \tau'' \to \tau'''$
and $\Sigma \vdash K' : \tau''' \to \tau$ Inversion on $\Sigma \vdash K : \tau'' \to \tau$

$\Sigma \vdash \texttt{ctx}(F') = \Gamma'$ By construction

$\texttt{loopnest}(F') = L$ By construction

Via case analysis on $\lhd e\ \texttt{ok}$: either $e =$ some v or $e = ()$.

If $e = v$ then

$\Sigma; \Gamma' \vdash v : \tau'$ Rule CHECKEXP-VALUE

$\Gamma \vdash v : A' \to A'$ Rule CHECKASSIGN-VALUE

$L \vdash v\ \texttt{ok}$ — Rule CHECKLOOP-VALUE

Otherwise $e = ()$

$\tau' = ()$ — Inversion on $\Sigma; \Gamma \vdash e : \tau'$

$\Sigma; \Gamma' \vdash () : \tau'$ — Rule CHECKEXP-TUPLE

$\Gamma \vdash () : A' \to A'$ — Rule CHECKASSIGN-TUPLE

$L \vdash ()\ \texttt{ok}$ — Rule CHECKLOOP-TUPLE

$\tau'' \neq \mathbf{cmd}$ — No derivation of $\mathbf{cmd}$ `small`

$\tau'' = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash e\ \texttt{canreturn}\ \tau'''$ — Vacuously true

$\Sigma \vdash K'; F' \lhd e : \tau$ — Rule CHECKSTATE-NORMAL

$\mu \mid K; F \rhd \texttt{loop}(e_c, e) \to \mu \mid K; F, \texttt{loopctx}(e_c, e) \rhd \texttt{if}(e_c, e, \texttt{break})$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\Sigma; \Gamma \vdash \texttt{loop}(e_c, e) : \mathbf{cmd}$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\texttt{assigned}(F) = A$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\Gamma \vdash \texttt{loop}(e_c, e) : A \to A$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\texttt{loopnest}(F) = L$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$L \vdash \texttt{loop}(e_c, e)\ \texttt{ok}$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{loop}(e_c, e)\ \texttt{canreturn}\ \tau''$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$

$\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$ — Rule CHECKFRAMETYPE-FRAMEEXPNORET

$\Sigma \vdash \texttt{ctx}(F, \texttt{loopctx}(e_c, e)) = \Gamma$ — Rule GETFRAMECONTEXT-LOOP

$\Sigma; \Gamma \vdash e_c : \mathbf{bool}$ — Inversion on $\Sigma; \Gamma \vdash \texttt{loop}(e_c, e) : \mathbf{cmd}$

$\Sigma; \Gamma \vdash e : \mathbf{cmd}$ — Inversion on $\Sigma; \Gamma \vdash \texttt{loop}(e_c, e) : \mathbf{cmd}$

$\Sigma; \Gamma \vdash \texttt{break} : \mathbf{cmd}$ — Rule CHECKEXP-BREAK

$\Sigma; \Gamma \vdash \texttt{if}(e_c, e, \texttt{break}) : \mathbf{cmd}$ — Rule CHECKEXP-IF

$\texttt{assigned}(F, \texttt{loopctx}(e_c, e)) = A$ — Rule GETFRAMEASSIGNED-LOOP

$\Gamma \vdash e_c : A \to A$ — Inversion on $\Gamma \vdash \texttt{loop}(e_c, e) : A \to A$

$\Gamma \vdash e : A \to A'$ — Inversion on $\Gamma \vdash \texttt{loop}(e_c, e) : A \to A$

$\Gamma \vdash \texttt{break} : A \to A$ — Rule CHECKASSIGN-BREAK

$A \cap A' = A$ — Lemma 25

$\Gamma \vdash \texttt{if}(e_c, e, \texttt{break}) : A \to A$ — Rule CHECKASSIGN-IF

$\texttt{loopnest}(F, \texttt{loopctx}(e_c, e)) = \texttt{inloop}$ — Rule GETLOOPCONTEXT-LOOP

$L \vdash e_c\ \texttt{ok}$ — Inversion on $L \vdash \texttt{loop}(e_c, e)\ \texttt{ok}$

$\texttt{inloop} \vdash e_c\ \texttt{ok}$ — Lemma 26

$\texttt{inloop} \vdash e\ \texttt{ok}$ — Inversion on $L \vdash \texttt{loop}(e_c, e)\ \texttt{ok}$

$\texttt{inloop} \vdash \texttt{break}\ \texttt{ok}$ — Rule CHECKLOOP-BREAK

$\texttt{inloop} \vdash \texttt{if}(e_c, e, \texttt{break})\ \texttt{ok}$ — Rule CHECKLOOP-IF

$\triangleright$ $\texttt{if}(e_c, e, \texttt{break})$ $\texttt{ok}$ — Rule DIROK-PUSHING

$\Sigma; \Gamma \vdash \texttt{loop}(e_c, e)$ $\texttt{canreturn}$ τ'' — Modus ponens with $\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{loop}(e_c, e)$ $\texttt{canreturn}$ τ''

$\Sigma; \Gamma \vdash e$ $\texttt{canreturn}$ τ'' — Inversion on $\Sigma; \Gamma \vdash \texttt{loop}(e_c, e)$ $\texttt{canreturn}$ τ''

$\Sigma; \Gamma \vdash \texttt{break}$ $\texttt{canreturn}$ τ'' — Rule ONLYRETURNS-BREAK

$\Sigma; \Gamma \vdash \texttt{if}(e_c, e, \texttt{break})$ $\texttt{canreturn}$ τ'' — Rule ONLYRETURNS-IF

$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{if}(e_c, e, \texttt{break})$ $\texttt{canreturn}$ τ'' — Trivial weakening

$\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleright \texttt{if}(e_c, e, \texttt{break}) : \tau$ — Rule CHECKSTATE-NORMAL

$\mu \mid K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{nop} \to \mu \mid K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{continue}$

$\texttt{loopnest}(F, \texttt{loopctx}(e_c, e)) = \texttt{inloop}$ — Rule GETLOOPCONTEXT-LOOP

$\texttt{innerloop}(F, \texttt{loopctx}(e_c, e)) = F, \texttt{loopctx}(e_c, e)$ — Rule GETINNERLOOP-LOOP

$\texttt{assigned}(F, \texttt{loopctx}(e_c, e)) = A$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{nop} : \tau$

$\Gamma \vdash \texttt{nop} : A \to A$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{nop} : \tau$

$\Sigma \vdash \texttt{ctx}(F, \texttt{loopctx}(e_c, e)) = \Gamma$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{nop} : \tau$

$\Sigma; \Gamma \vdash \texttt{nop} : \mathbf{cmd}$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{nop} : \tau$

$\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{nop} : \tau$

$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{nop} : \tau$

$\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \triangleleft \texttt{continue} : \tau$ — Rule CHECKSTATE-LOOPCONT

$\mu \mid K; F \triangleright \texttt{break} \to \mu \mid K; F \triangleleft \texttt{break}$

$\texttt{inloop} \vdash \texttt{break}$ $\texttt{ok}$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\texttt{assigned}(F) = A$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{break} : A \to \{x_1, \ldots, x_n\}$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\Sigma; \Gamma \vdash \texttt{break} : \tau'$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\tau' = \mathbf{cmd}$ — Inversion on $\Sigma; \Gamma \vdash \texttt{break} : \tau'$

$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F \triangleright \texttt{break} : \tau$

$\texttt{innerloop}(F) = F', \texttt{loopctx}(e_c, e)$ and $\texttt{assigned}(F') = A'$ and

$\Sigma; A' \vdash F' : \mathbf{cmd} \to \tau''$ — Lemma 12

$\Sigma; A' \vdash F', \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$ — Rule CHECKFRAMETYPE-FRAMELOOP

$\Sigma \vdash K; F \triangleleft \texttt{break} : \tau$ — Rule CHECKSTATE-LOOPBRK

$\mu \mid K; F, x : \tau_x = v \triangleleft \texttt{break} \to \mu \mid K; F \triangleleft \texttt{break}$

$\texttt{assigned}(F') = A$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \ : \tau$
$\Sigma; A \vdash F' : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \ : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \ : \tau$
$\texttt{loopnest}(F, x : \tau_x = v) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \ : \tau$
$\texttt{innerloop}(F, x : \tau_x = v) = F'$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \ : \tau$
$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\texttt{loopnest}(F, x : \tau_x = v) = \texttt{inloop}$
$\texttt{innerloop}(F) = F'$ — Inversion on $\texttt{innerloop}(F, x : \tau_x = v) = F'$
$\Sigma \vdash K; F \lhd \texttt{break} : \tau$ — Rule CHECKSTATE-LOOPBRK

$\mu \mid K; F, x : \tau \lhd \texttt{break} \to \mu \mid K; F \lhd \texttt{break}$

$\texttt{assigned}(F') = A$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{break} : \tau$
$\Sigma; A \vdash F' : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{break} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{break} : \tau$
$\texttt{loopnest}(F, x : \tau_x) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{break} : \tau$
$\texttt{innerloop}(F, x : \tau_x) = F'$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{break} : \tau$
$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\texttt{loopnest}(F, x : \tau_x) = \texttt{inloop}$
$\texttt{innerloop}(F) = F'$ — Inversion on $\texttt{innerloop}(F, x : \tau_x) = F'$
$\Sigma \vdash K; F \lhd \texttt{break} : \tau$ — Rule CHECKSTATE-LOOPBRK

$\mu \mid K; F, e \lhd \texttt{break} \to \mu \mid K; F \lhd \texttt{break}$

$\texttt{assigned}(F') = A$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{break} : \tau$
$\Sigma; A \vdash F' : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{break} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{break} : \tau$
$\texttt{loopnest}(F, e) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{break} : \tau$
$\texttt{innerloop}(F, e) = F'$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{break} : \tau$
$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\texttt{loopnest}(F, e) = \texttt{inloop}$
$\texttt{innerloop}(F) = F'$ — Inversion on $\texttt{innerloop}(F, e) = F'$
$\Sigma \vdash K; F \lhd \texttt{break} : \tau$ — Rule CHECKSTATE-LOOPBRK

$\mu \mid K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{break} \to \mu \mid K; F \lhd \texttt{nop}$

$\texttt{innerloop}(F, \texttt{loopctx}(e_c, e)) = F'$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{break} : \tau$
$F' = F, \texttt{loopctx}(e_c, e)$ — Inversion on $\texttt{innerloop}(F, \texttt{loopctx}(e_c, e)) = F'$
$\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{break} : \tau$
$\texttt{assigned}(F) = A$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{break} : \tau$

$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ for some Γ — By construction
$\Sigma \vdash \texttt{nop} : \mathbf{cmd}$ — Rule CHECKVAL-NOP
$\Sigma; \Gamma \vdash \texttt{nop} : \mathbf{cmd}$ — Rule CHECKEXP-VALUE
$\Gamma \vdash \texttt{nop} : A \to A$ — Rule CHECKASSIGN-VALUE
$\texttt{loopnest}(F) = L$ for some L — By construction
$L \vdash \texttt{nop ok}$ — Rule CHECKLOOP-VALUE
$\lhd \texttt{nop ok}$ — Rule DIROK-RETURNING
$\Sigma; \Gamma \vdash \texttt{nop canreturn } \tau''$ — Rule ONLYRETURNS-NOP
$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{nop canreturn } \tau''$ — Always true
$\Sigma \vdash K; F \lhd \texttt{nop} : \tau$ — Rule CHECKSTATE-NORMAL

$\mu \mid K; F \rhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}$

$\texttt{inloop} \vdash \texttt{continue ok}$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\texttt{assigned}(F) = A$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\cdot, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \texttt{continue} : A \to \{x_1, \ldots, x_n\}$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\Sigma; \Gamma \vdash \texttt{continue} : \tau'$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\tau' = \mathbf{cmd}$ — Inversion on $\Sigma; \Gamma \vdash \texttt{continue} : \tau'$
$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F \rhd \texttt{continue} : \tau$
$\texttt{innerloop}(F) = F', \texttt{loopctx}(e_c, e)$ and $\texttt{assigned}(F') = A'$ and
$\Sigma; A' \vdash F' : \mathbf{cmd} \to \tau''$ — Lemma 12
$\Sigma; A' \vdash F', \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$ — Rule CHECKFRAMETYPE-FRAMELOOP
$\Sigma \vdash K; F \lhd \texttt{continue} : \tau$ — Rule CHECKSTATE-LOOPCONT

$\mu \mid K; F, x : \tau_x = v \lhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}$

$\texttt{assigned}(F') = A$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{continue} : \tau$
$\Sigma; A \vdash F' : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{continue} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{continue} : \tau$
$\texttt{loopnest}(F, x : \tau_x = v) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{continue} : \tau$
$\texttt{innerloop}(F, x : \tau_x = v) = F'$ — Inversion on $\Sigma \vdash K; F, x : \tau_x = v \lhd \texttt{continue} : \tau$
$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\texttt{loopnest}(F, x : \tau_x = v) = \texttt{inloop}$
$\texttt{innerloop}(F) = F'$ — Inversion on $\texttt{innerloop}(F, x : \tau_x = v) = F'$
$\Sigma \vdash K; F \lhd \texttt{continue} : \tau$ — Rule CHECKSTATE-LOOPCONT

$\mu \mid K; F, x : \tau \lhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}$

$\texttt{assigned}(F') = A$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{continue} : \tau$
$\Sigma; A \vdash F' : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{continue} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{continue} : \tau$
$\texttt{loopnest}(F, x : \tau_x) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{continue} : \tau$
$\texttt{innerloop}(F, x : \tau_x) = F'$ — Inversion on $\Sigma \vdash K; F, x : \tau_x \lhd \texttt{continue} : \tau$
$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\texttt{loopnest}(F, x : \tau_x) = \texttt{inloop}$
$\texttt{innerloop}(F) = F'$ — Inversion on $\texttt{innerloop}(F, x : \tau_x) = F'$
$\Sigma \vdash K; F \lhd \texttt{continue} : \tau$ — Rule CHECKSTATE-LOOPCONT

$\mu \mid K; F, e \lhd \texttt{continue} \to \mu \mid K; F \lhd \texttt{continue}$

$\texttt{assigned}(F') = A$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{continue} : \tau$
$\Sigma; A \vdash F' : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{continue} : \tau$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{continue} : \tau$
$\texttt{loopnest}(F, e) = \texttt{inloop}$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{continue} : \tau$
$\texttt{innerloop}(F, e) = F'$ — Inversion on $\Sigma \vdash K; F, e \lhd \texttt{continue} : \tau$
$\texttt{loopnest}(F) = \texttt{inloop}$ — Inversion on $\texttt{loopnest}(F, e) = \texttt{inloop}$
$\texttt{innerloop}(F) = F'$ — Inversion on $\texttt{innerloop}(F, e) = F'$
$\Sigma \vdash K; F \lhd \texttt{continue} : \tau$ — Rule CHECKSTATE-LOOPCONT

$\mu \mid K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{continue} \to \mu \mid K; F \rhd \texttt{loop}(e_c, e)$

$\texttt{innerloop}(F, \texttt{loopctx}(e_c, e)) = F'$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{continue} : \tau$
$F' = F, \texttt{loopctx}(e_c, e)$ — Inversion on $\texttt{innerloop}(F, \texttt{loopctx}(e_c, e)) = F'$
$\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{continue} : \tau$
$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\texttt{assigned}(F, \texttt{loopctx}(e_c, e)) = A$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\texttt{assigned}(F) = A$ — Inversion on $\texttt{assigned}(F, \texttt{loopctx}(e_c, e)) = A$
$\Sigma; A \vdash F : \mathbf{cmd} \to \tau''$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\Sigma \vdash K : \tau'' \to \tau$ — Inversion on $\Sigma \vdash K; F, \texttt{loopctx}(e_c, e) \lhd \texttt{continue} : \tau$
$\Sigma \vdash \texttt{ctx}(F) = \Gamma$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\Sigma; \Gamma \vdash \texttt{loop}(e_c, e) : \mathbf{cmd}$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\Gamma \vdash \texttt{loop}(e_c, e) : A' \to A'$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\texttt{loopnest}(F) = L$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$L \vdash \texttt{loop}(e_c, e)\ \texttt{ok}$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\rhd \texttt{loop}(e_c, e)\ \texttt{ok}$ — Rule DIROK-PUSHING
$\Sigma; \Gamma \vdash \texttt{loop}(e_c, e)\ \texttt{canreturn}\ \tau''$ — Inversion on $\Sigma; A \vdash F, \texttt{loopctx}(e_c, e) : \mathbf{cmd} \to \tau''$
$\mathbf{cmd} = \mathbf{cmd} \Rightarrow \Sigma; \Gamma \vdash \texttt{loop}(e_c, e)\ \texttt{canreturn}\ \tau''$ — Always true

$$\Sigma \vdash K; F \rhd \texttt{loop}(e_c, e) : \tau$$ Rule CHECKSTATE-NORMAL

$$\frac{}{\mu \mid K; F \lhd \texttt{exn} \to \mu \mid \cdot \lhd \texttt{exn}}$$

$$\Sigma \vdash \cdot \lhd \texttt{exn} : \tau$$ Rule CHECKSTATE-EXN

Appendix B

Standard Libraries

These are the standard libraries as of this publication.

B.1 Input/Output

B.1.1 `conio`

The `conio` library contains functions for performing basic console input and output.

```
void print(string s)
```

Prints `s` to standard output.

```
void println(string s)
```

Same as `print` but unconditionally prints a newline `\n`.

```
void printint(int i)
```

A simple convenience function which converts `i` to a string and calls `print`.

```
string readline()
```

Parses a newline (`\n` or `\r\n`) delimited sequence of characters from standard input and returns them as a string.

B.1.2 `file`

The `file` library contains functions for reading lines out of files. Files handles are represented by the `file_t` type. The handle contains an internal position which ranges from 0 to the logical

size of the file in bytes. File handles should be closed when they are no longer needed. The program must close them explicitly - garbage collection of the file handle will not close it.

```
file_t file_read(string path)
```

Creates a handle for reading from the file given by the specified path. If the file cannot be opened for reading, the program aborts.

```
void file_close(file_t f)
```

Releases any resources associated with the file handle. This function should not be invoked twice on the same handle.

```
bool file_eof(file_t f)
```

Returns true if the internal position of the handle is the size of the file.

```
string file_readline(file_t f)
```

Reads a line from the given file, advancing the handle's internal position by the number of characters in the returned string plus the delimeters. Lines are denoted by the `\n` or `\r\n` characters or the end of the file.

B.1.3 `args`

The `args` library provides functions for basic argument parsing. There are several functions that set up the description of the argument schema and then a single function (`args_parse`) which performs the parsing.

```
void args_flag(string name, bool *ptr)
```

Describes a simple boolean flag. If present on the command line, `args_parse` sets `*ptr` to **`true`**.

```
void args_int(string name, int *ptr)
```

Describes a switch expecting an integer of the form accepted by `parse_integer` with `base = 0`. If present on the command line,
`args_parse` sets `*ptr` to the value parsed from the switch. If the value could not be parsed, it is not set.

```
void args_string(string name, string *ptr)
```

Describes a switch expecting some additional argument. If present on the command line, `args_parse` sets `*ptr` to the argument.

```
string[] args_parse()
```

Attempts to parse the command line arguments given to the program by the operating system. Arguments that indicate a switch consume the next argument. Arguments that are not matched to switches or flags are considered positional arguments and are returned in an array.

B.2 Data manipulation

B.2.1 `parse`

The `parse` library provides two functions to parse integers and booleans. These functions return pointers to two structs: **`struct`** `parsed_bool` and **`struct`** `parsed_int`.
`struct` `parsed_bool` is has the following members:

`bool` `parsed`	Indicates if the string was successfully parsed
`bool` `value`	If the `parsed` field is true, holds the parsed value

`struct` `parsed_int` has the same members except that `value` is of type **`int`**.

```
struct parsed_bool *parse_bool(string s)
```

Attempts to parse `s` into a value of type `bool`. Accepts `"true"` and `"false"` as valid strings.

```
struct parsed_int *parse_int(string s, int b)
```

Attempts to parse `s` as a number written in base `b`. Supported bases include 8, 10 and 16. If `b` is 0, the base of the number is inferred from the leading digits. `0x` indicates that the base is 16, otherwise `0` indicates base 8 and any other digit indicates base 10.

If the number is too large to be represented as an **`int`**, number is not parsed.

B.2.2 `string`

The `string` library contains a few basic routines for working with strings and characters.

```
int string_length(string s)
```

Returns the number of characters in `s`.

```
char string_charat(string s, int n)
```

Returns the `n`th character in `s`. If `n` is less than zero or greater than the length of the string, the program aborts.

```
string string_join(string a, string b)
```

Returns a string containing the contents of `b` appended to the contents of `a`.

```
string string_sub(string s, int start, int end)
```

Returns the substring composed of the characters of `s` beginning at index given by `start` and continuing up to but not including the index given by end If `end <= start`, the empty string is returned If `end < 0` or `end >` the length of the string, it is treated as though it were equal to the length of the string. If `start < 0` the empty string is returned.

```
bool string_equal(string a, string b)
```

Returns **`true`** if the contents of `a` and `b` are equal and **`false`** otherwise.

```
int string_compare(string a, string b)
```

Compares `a` and `b` lexicographically. If `a` comes before `b`, then the return value is negative. If `string_equal(a,b)` is **`true`**, the return value is 0. Otherwise the return value is greater than 0.

```
bool char_equal(char a, char b)
```

Returns whether or not the two characters are identical.

```
int char_compare(char a, char b)
```

Compares the two characters according to their ASCII encoding. If the two characters have the same encoding, the return value is 0. If `char_ord(a) < char_ord(b)`, then the return value is less than 0. Otherwise it is greater than 0.

```
string string_frombool(bool b)
```

Returns a canonical representation of `b` as a string. The returned value will always be parsed by `parse_bool` into a value equal to `b`.

```
string string_fromint(int i)
```

Returns a canonical representation of `i` as a string. The returned value will always be parsed by `parse_int` into a value equal to `i`.

```
string string_tolower(string s)
```

Returns a string containing the same character sequence as `s` but with each upper case character replaced by its lower case version.

```
char[] string_to_chararray(string s)
```

Returns the characters in `s` as an array. The length of the array is one more than the length of `s`. The last character in the array is `'\0'`.

```
string string_from_chararray(char[] A)
```

Returns a string containing the characters from `A` in it. The last character of the array must be equal to `'\0'`. The program will abort if it is not.

```
int char_ord(char c)
```

Returns an integer representing the ASCII encoding of `c`.

```
char char_chr(int n)
```

Decodes `n` as an ASCII character and returns the result. If `n` cannot be decoded as valid ASCII, the program aborts.

B.3 Images

B.3.1 `img`

The `img` library defines a type for two dimensional images represented as pixels with 4 color channels - alpha, red, green and blue - packed into one `int`. It defines an image type `image_t`. Images must be explicitly destroyed when they are no longer needed with the `image_destroy` function.

```
image_t image_create(int width, int height)
```

Creates an image with the given width and height. The default pixel color is transparent black. `width` and `height` must be positive.

```
image_t image_clone(image_t image)
```

Creates a copy of the image.

```
void image_destroy(image_t image)
```

Releases any internal resources associated with `image`. The array returned by a previous `image_data` call will remain valid however any subsequent calls using `image` will cause the program to abort.

```
image_t image_subimage(image_t image, int x, int y, int w, int h)
```

Creates a partial copy of `image` using the rectangle as the source coordinates in `image`. Any parts of the given rectangle that are not contained in `image` are treated as transparent black.

```
image_t image_load(string path)
```

Loads an image from the file given by `path` and converts it if need be to an ARGB image. If the file cannot be found, the program aborts.

```
void image_save(image_t image, string path)
```

Saves `image` to the file given by `path`. If the file cannot be written, the program aborts.

```
int image_width(image_t image)
```

Returns the width in pixels of `image`.

```
int image_height(image_t image)
```

Returns the height in pixels of `image`.

```
int[] image_data(image_t image)
```

Returns an array of pixels representing the image. The pixels are given line by line so a pixel at (x,y) would be located at `y*image_width(image) + x`. Any writes to the array will be reflected in calls to `image_save`, `image_clone` and `image_subimage`. The channels are encoded as `0xAARRGGBB`.

Appendix C

Code Listing

C.1 Sample `c0defs.h`

```
// Defines bool, true, and false
#include <stdbool.h>

typedef const char *string;

#define alloc(ty) ((ty*)calloc(1, sizeof (ty)))
#define alloc_array(ty, size) ((ty*)calloc(size, sizeof (ty)))
```

C.2 C_0 runtime interface

```
#include <stddef.h>

typedef struct c0_array c0_array;

// Aborts execution and notifies the user of the reason
C0API void c0_abort(const char *reason);

// Allocates from the GC heap
C0API void *c0_alloc(size_t bytes);
// Allocate an array of elemsize elements of elemcount bytes per element
C0API c0_array *c0_array_alloc(size_t elemsize, int elemcount);

// Returns a pointer to the element at the given index of the given array
// Runtimes may ignore the element size
C0API void *c0_array_sub(c0_array *a, int idx, size_t elemsize);

#ifdef C0_HAVE_CONCRETE_RUNTIME
// Returns the length of the array. This is only permitted in certain C0
// programs since not all runtimes may support it.
C0API int c0_array_length(c0_array *a);
```

```
#endif

// Returns the empty string
C0API c0_string c0_string_empty();
// Returns the length of the given string
C0API int c0_string_length(c0_string s);

// Returns the character at the given index of the string. If the
// index is out of range, aborts.
C0API c0_char c0_string_charat(c0_string s, int i);

// Returns the substring composed of the characters of s beginning at
// index given by start and continuing up to but not including the
// index given by end
// If end <= start, the empty string is returned
// If end < 0 or end > the length of the string, it is treated as though
// it were equal to the length of the string.
// If start < 0 the empty string is returned.
C0API c0_string c0_string_sub(c0_string s, int a, int b);

// Returns a new string that is the result of concatenating b to a.
C0API c0_string c0_string_join(c0_string a, c0_string b);

// Constructs a c0rt_string_t from a null-terminated string
C0API c0_string c0_string_fromcstr(const char *s);

// Returns a null-terminated string from a c0_string. This string must
// be explicitly deallocated by calling c0_string_freecstr.
C0API const char *c0_string_tocstr(c0_string s);

// Frees a string returned by c0_string_tocstr
C0API void c0_string_freecstr(const char *s);

// Returns a c0_string from a string literal.
C0API c0_string c0_string_fromliteral(const char *s);

C0API bool c0_string_equal(c0_string a, c0_string b);
C0API int c0_string_compare(c0_string a, c0_string b);
C0API bool c0_char_equal(c0_char a, c0_char b);
C0API int c0_char_compare(c0_char a, c0_char b);
```

Bibliography

[1] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, June 2010. 2.9

[2] Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: an alternative to strings. *Software Practice & Experience*, 25(12):1315–1330, December 1995. 4.2

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, 18(9):807–820, 1988. 4.2

[4] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM. 6.1

[5] ECMA. *ECMA-367: Eiffel analysis, design and programming language*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, June 2006. URL `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-367.p` 2006. 2.9

[6] Go. The Go programming language. URL `http://golang.org/doc/go_spec.html`. 6.1, 6.2.6

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005. 1.2.1

[8] Dan Grossman. Quantified types in an imperative language. *ACM Trans. Program. Lang. Syst.*, 28(3):429–475, 2006. 2.10.2

[9] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of C. *C/C++ User's Journal*, 23(1), January 2005. 2.10.2

[10] Peter B. Henderson, Thomas J. Cortina, and Jeannette M. Wing. Computational thinking. *SIGCSE Bull.*, 39(1):195–196, 2007. URL `http://dx.doi.org/10.1145/1227504.1227378`. 1.1

[11] ISO. ISO C standard 1999. Technical report, 1999. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`. ISO/IEC 9899:1999 draft. 1.2.1

[12] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005. URL

ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf. 2.9

[13] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, rev sub edition, May 1997. URL http://www.worldcat.org/isbn/0262631814. 1.2.1

[14] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code, 2002. 6.1

[15] PrJavaLibs. Standard libraries. URL http://www.cs.princeton.edu/introcs/stdlib/. 5.2

[16] Casey Reas and Benjamin Fry. Processing.org: programming for artists and designers. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Web graphics*, page 3, New York, NY, USA, 2004. ACM. 5.2, 6.1