A Formal Semantics of C with Applications

Technical Report

Chucky Ellison Grigore Roşu
University of Illinois
{celliso2, grosu}@illinois.edu

Abstract

This paper describes an executable formal semantics of C expressed using a formalism based on term rewriting. Being executable, the semantics has been thoroughly tested against the GCC torture test suite and successfully passes over 96% of 715 test programs. It is the most complete and thoroughly tested formal definition of C to date.

The semantics yields an interpreter, debugger, and state space search tool "for free". The semantics is shown capable of automatically finding program errors, both statically and at runtime. It is also used to enumerate nondeterministic behavior. These techniques together allow the tool to identify undefined programs.

The entire C semantics is included as Appendix B.

1. Introduction

C is one of the most frequently used programming languages. It provides just enough abstraction above assembly language for programmers to get their work done without having to worry about the details of the machines on which the programs run. Despite this abstraction, C is also known for the ease in which it allows programmers to write buggy programs. With no runtime checks, and little static checking, in C the programmer is to be trusted entirely. Despite the abstraction, the language is still low-level enough that programmers *can* take advantage of assumptions about the underlying architecture. Trust in the programmer and the ability to write *non*-portable code are actually two of the design principles under which the C standard was written [20]. These ideas often work in concert to yield intricate, platform-dependent bugs. The potential subtlety of C bugs makes it an excellent candidate for formalization, as subtle bugs can often be caught only by more rigorous means.

In this paper, we present a formal semantics for C that can be used for finding program bugs. Rather than being an "on paper" semantics, the definition is written in an executable, machine-readable form and has been tested against the GCC torture tests. We report on this evaluation in Section 7. The semantics is based on the ISO/IEC 9899:1999 (C99) standard [19]. Where appropriate, we reference the forthcoming ISO/IEC 9899:201x (C1X) standard [21], which supersedes the C99 standard by addressing problems and providing clarifications and extensions. We are handling some of the extensions to C provided in C99 such as flexible array members, but not features such as variable length arrays or complex number support. We discuss the precise feature set later in this introduction.

The standard broadly categorizes the particular behaviors of any C implementation into four categories: unspecified, implementation-defined, undefined, and locale-specific behavior. For the purposes of this paper, we focus on only three of these [21, §3.4]:

unspecified behavior Use of an unspecified value, or other behavior [with] two or more possibilities and [. . .] no further requirements on which is chosen in any instance.

implementation-defined Unspecified behavior where each implementation documents how the choice is made.

undefined behavior Behavior, upon use of a non-portable or erroneous program construct or of erroneous data, [with] no requirements.

An example of unspecified behavior is the order in which the arguments to a function are evaluated. An example of implementation defined behavior is the size of an **int**. An example of undefined behavior is referring to an object outside of its lifetime.

To put these definitions in perspective, for a C program to be maximally portable, "it shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit" [21, §4.5]. This is called "strict conformance". However, many C programs are inherently non-portable (e.g., device drivers). The standard offers another level of conformance (simply called "conforming") where the program may rely on particular implementation-defined or even unspecified (but never undefined) behavior. Because the standard requires that all implementation-defined behaviors are documented by an implementation, but does not require this for unspecified behaviors, we decided as a rule to give particular implementation-defined behaviors to our semantics, and only when necessary for common programs to also describe unspecified behaviors. We do not give semantics to any undefined behavior. As much as possible, this behavior is kept separate from the semantics underlying the high-level (defined for all implementations) aspects of the language. More details about this parameterization is described in Section 4.5.4.

This specification of low-level detail, coupled with the use of an operational semantic framework, gives us an executable semantics. Above all else, our semantics has been motivated by the desire to develop formal, yet practical tools for C. Few languages are designed using formal specifications. When C was being standardized, they explored using formal methods, but in the end they decided to use simple prose because, "Anything more ambitious was considered to be likely to delay the Standard, and to make it less accessible to its audience" [20, §6]. We developed our semantics in roughly 6 person – months. To put this in perspective, one member of the standards committee estimated that it took roughly 62 person – years to produce the C99 standard [22]. We are *not* claiming that we have done the same job in a fraction of the time (we have done a fraction of the job in a fraction of the time). We are only pointing out that the development of a formal semantics could have taken place alongside the development of the standard. For formal methods to be accepted by the general language community, it needs to be shown to have value beyond that of prose. We developed our semantics in such a way so that the single definition could be used immediately for interpreting, debugging, or analysis (described in Section 6).

However, this executability does not mean that our definition is not formal. Being written in rewriting logic, it comes with a complete proof system and initial model semantics [25]. Briefly, a rewrite system is a set of rules over terms constructed from a signature. The rewrite rules match and apply everywhere, making rewriting logic a simple, uniform, and general formal computational paradigm. This is explained in greater detail in Section 4.1.

Our C semantics defines approximately 120 C syntactic operators and 200 intermediate or auxiliary semantic operators. The definitions of these operators are given by 400 semantic rules and 172 helper rules (such as **sizeof**(**int**) $\Rightarrow$ 4) spread over 2,333 source lines of code (SLOC). However, it takes only 37 of those rules (201 SLOC) to cover the behavior of statements, and another 119 for expressions (417 SLOC). There are 353 rules for dealing with types, memory, and other necessary mechanisms. Finally, there are about 63 rules for the core of our standard library. Our semantics is described in more detail in Section 4, while the entire semantics can be found in Appendix B. Additionally, the machine readable source can be found at `https://code.google.com/p/k-framework/wiki/DefinitionOfC`.

Contributions The specific contributions of this paper include:

- a detailed comparison of other C formalisms;
- the most comprehensive formal semantics of C to date, which is executable and has been thoroughly tested;
- and demonstrations as to its utility in discovering program flaws.

Features Unless otherwise specified, all aspects related to the below features are included and are given a direct semantics (not by a translation to other features):

- Expressions: referencing and dereferencing, casts, array indexing (`a[i]`), structure members (`->` and `.`), arithmetic, bitwise, and logical operators, sizeof, increment and decrement, assignments, sequencing[1] (`_,_`), ternary conditional[1] (`_?_:_`)
- Statements: for,[1] do-while,[1] while, if, if/else, switch, goto, break, continue, return
- Types and Declarations: enums, structs, unions, bitfields, initializers, static storage,[1] typedefs
- Values: regular scalar values (signed/unsigned arithmetic and pointer types), structs, unions
- Standard Library: malloc/free, set/longjmp, basic I/O
- Environment: command line arguments
- Conversions: (implicit) argument and parameter promotions and arithmetic conversion, and (explicit) casts

2. Comparison with Existing Formal C Semantics

There have already been a number of formal semantics written for C. One might (rightfully) ask, "Why yet another?" We claim that the definitions so far have either made enough simplifying assumptions that for many purposes they are not C, or have lacked any way to use them other than on paper. While "paper semantics" are useful for teaching and understanding the language, we believe that without a mechanized definition, it is difficult to gain confidence in a definition's appropriateness for any other purpose. Below we highlight the most prominent definitions and explain their successes and shortcomings in comparison with our work.

Gurevich and Huggins (GH) One of the earliest formal descriptions of ANSI C is given by Gurevich and Huggins [17], using abstract state machines (ASMs) (then known as evolving algebras). Their semantics describes C using four increasingly precise layers, each formal and analyzable. Their semantics covers all the high-level constructs of the language, and uses external oracles to capture the underspecification inherent in the definition of C. Their semantics was written without access to a standard, and so is based on Kernighan and Ritchie [24]. However, many behavioral details of the lowest-level features of C are now partially standardized, including details of arithmetic, type representation, and evaluation strategies. The latter has been investigated in the context of ASMs [40], but none are present in the original definition. Based on our own experience, the details involving the lowest-level features of C are incredibly complex (see Section 3), but we see no reason why the ASM technique could not be used to specify them.

Their semantics was never converted into an executable tool, nor has it been used in applications. However, their purpose and context was different from ours. As pointed out elsewhere [29, p. 11], their semantics was constructed without the benefit of any mechanization. According to Gurevich,[2] their purpose was to "discover the structure of C," at a time when "C was far beyond the reach of denotational semantics, algebraic specifications, etc."

Cook, Cohen, and Redmond (CCR) Soon after the previous definition, Cook et al. [7] describe a denotational semantics of C90 using a custom-made temporal logic for the express purpose of proving properties about C programs. Their grammar covers nearly the entire C syntax, although they desugar some of it into other constructs to reduce the number of primitives that need independent semantics. We effectively do the same thing by using CIL (see Section 4.2). Also like us, they give semantics for particular implementation-defined behaviors in order to have a more concrete definition. These choices are then partitioned off so that one could, in theory, choose different implementation-defined values and behaviors.

They have given at least a basic semantics to most C constructs. We say "at least" without malicious intent—although their work was promising, they moved on to other projects before developing a testable version of their semantics and without doing any concrete evaluation.[2] Additionally, no proofs were done using this semantics.

Cook and Subramanian (CS) The related work of Cook and Subramanian [6, 38] is a semantics for a restricted subset of C, based loosely on the semantics above. This semantics is embedded in the theorem prover Nqthm [4] (a precursor to ACL2). They were successful in verifying at least two functions: one that takes two pointers and swaps the values at each, and one that computes the factorial. They were also able to prove properties about the C definition itself. For example, they prove that the execution of `p = &a[n]` puts the address of the nth element of the array `a` into `p` [6, p. 122]. Their semantics is, at its roots, an interpreter—it uses a similar technique to that described by Blazy and Leroy [2] to coax an interpreter from recursive functions—but there is no description in their work of any reference programs they were capable of executing. As above, it appears the work was terminated before it was able to blossom.

Norrish (No) A more recent definition is presented by Norrish [29], who gives both static and dynamic formal semantics inside the HOL theorem proving system for the purpose of verifying C programs (later extended to C++ [30]). His semantics is in the SOS style, using small-step for expressions and big-step for statements. One of the focuses of his work is to present a precise description of the allowable evaluation orders of expressions. To date, his semantics still stands as the most precise representation of evaluation. In Section 6.3 we demonstrate how our definition captures the same behaviors.

Working inside HOL provides an elegant solution to the underspecification of the standard—he can state facts given by the standard as axioms/theorems. To maintain executability, we chose instead to parameterize the definition for those semantic choices

[1] These features are handled implicitly by C Intermediate Language (CIL).

[2] Personal communication, 2010.

that are implementation-defined. In that respect, our definitions conceptually complement each other—his is better for formal proofs about C, while ours is better for searching for behaviors in C programs (see Section 6.3.1). Proofs of program correctness [36] as well as semantics-level proofs [11] have already been demonstrated in the framework used by our semantics, but we have not yet applied these techniques to C.

Norrish uses his definition to prove some properties about C itself, as well as to verify some strong properties of simple ($\leq$ 5 line) programs, but was unable to apply his work to larger programs. His semantics is not executable, so it has not been tested against actual programs. However, the proofs done within the HOL system help lend confidence to the definition.

Papaspyrou (Pa) A denotational semantics for C99 is described by Papaspyrou [31, 32] using a monadic approach to domain construction. The definition includes static, typing, and dynamic semantics, which enables him not only to represent the behavior of executing programs, but also check for errors like redefinition of an identifier in the same scope. Papaspyrou, Norrish, and Cook et al. each give a typing semantics in addition to the dynamic semantics, while we and Blazy and Leroy (below) give only dynamic semantics.

Papaspyrou represents his semantics in Haskell, yielding a tool capable of searching for program behaviors. This was the only semantics for which we were able to obtain a working interpreter, and we were able to run it on a few examples. Having modeled expression non-determinism, and being denotational, his semantics evaluates a program into a set of possible return values. However, we found his interpreter to be of limited capability in practice. For example, using his definition, we were unable to compute the factorial of 6 or the fourth Fibonacci number.

Blazy and Leroy (BL) A big-step operational semantics for a subset of C is given by Blazy and Leroy [2]. While they do not claim to have given semantics for the entirety of C, their semantics does cover most of the major features of the language and has been used in a number of proofs including the verification of the optimizing compiler CompCert. Like us, they also use CIL as a front-end, but decided to patch it so that it would not transform loops, and therefore retain `while`, `do-while`, and `for` loops individually. For them this was a necessity, as CIL also transforms `continue` statements into `goto`s, which they do not handle.

To help validate the semantics, they have done manual reviews of the definition as well as proved properties of the semantics such as determinism of evaluation. They additionally have verified semantics-preserving transformations from their language into simpler languages, which are easier to develop confidence in. Their semantics is not directly executable, but they describe a mechanism by which they could create an equivalent recursive function that would act as an interpreter. This work has not yet been completed. They are also working on a small-step and an axiomatic semantics in order to prove relationships between the semantics—the small step semantics has been completed since publication, and is now being used in CompCert to handle `goto`.[3]

Their semantics does not handle sub-expressions with side effects, which, because their semantics is big-step, represents a significant barrier to obtaining a full semantics of C. Without changing most of their current rules, adding expression side effects (and the nondeterminism that comes along with that) would be a difficult undertaking. It is important to point out that writing a full semantics was not their intention—they deliberately made this decision to cover a subset of features and behavior to simplify their semantics and make corresponding proofs easier. Based on their success with the related formal proofs, this appears to have paid off.

[3] Personal communication, 2010.

	Definition						
Feature	GH	CCR	CR	No	Pa	BL	—
Bitfields	●	◐	○	○	◐	○	●
Enums	◐	●	○	○	●	○	●
Floats	○	○	○	○	●	●	◐
String Literal	○	●	○	○	●	●	●
Struct/Union	●	●	●	◐	●	●	●
Struct as Value	○	○	○	●	○	○	●
Arithmetic	◐	●	●	○	●	●	●
Bitwise	○	●	○	○	●	●	●
Casts	◐	◐	○	◐	◐	●	●
Functions	●	●	◐	●	●	●	●
Exp. Side Effects	●	●	○	●	●	○	●
Variadic Funcs.	○	○	○	○	○	○	●
Alignment	○	○	○	○	○	●	○
Eval. Strategies	○	◐	○	●	●	○	●
Overflow	○	○	○	○	○	○	●
Volatile	○	○	○	○	○	◐	○
Break/Continue	◐	●	◐	●	●	●	●
Goto	◐	○	○	○	●	○	●
Switch	◐	●	○	○	●	◐	●
Longjmp	○	○	○	○	○	○	●
Malloc	○	○	○	○	○	○	●
Feature	GH	CCR	CR	No	Pa	BL	—

●: Fully Described ◐: Partially Described ○: Not Described

GH represents Gurevich and Huggins [17], *CCR* is Cook et al. [7], *CR* is Cook and Subramanian [6], *No* is Norrish [29], *Pa* is Papaspyrou [32], *BL* is Blazy and Leroy [2], and — is our work.

Figure 1. Dynamic Semantics Features

There are other formal semantics of C (or fragments of C) that we choose not to review here, including Black [1] and Bofinger [3], as they either focus on subsets subsumed by the work previously discussed, or do not give dynamic semantics.

We condense our study of related works into a simple chart shown in Figure 1. For interested parties, this chart may be contentious. However, we believe that it is useful, both for developers of formal semantics of C and for users of them, to give a broad (though admittedly incomplete) overview of the state of the art of the formal semantics of C. Also, it may serve as an indication of the complexity involved in the C language. Note that not all features are equally difficult. Adding additional rules to catch arithmetic overflow would be much easier than adding support for bitfields, for example. Like the example above with a big-step semantics and different evaluation orders, adding new features can sometimes be difficult, depending on the semantical style or way in which the constructs were described.

We did our best to give the authors the benefit of the doubt with features they explicitly mentioned, but the other features were based on our reading of their semantics. We have also discussed our views with the authors, where possible, to try and establish a consensus. Obviously the categories are broad, but our intention is to give an overview of some of the more difficult features of C. We purposefully left off any feature that all definitions had fully defined.

Finally, there are a number of other emergent features, such as multi-dimensional arrays, that are difficult to discern correctness through simple inspection of the formal semantics (i.e., without testing or verifying it). It is also difficult to determine if feature pairs work together—for example, does a definition allow bitfields inside of unions? We decided to leave most of these features out of the

chart because they are simply too hard to determine if the semantics were complete enough for them to work properly.

3. Why Details Matter

It is tempting to gloss over the details of C's arithmetic and other low-level features when giving it a formal semantics. However, the language is designed to be translatable to common machine architectures where there are particular instructions for adding 16-bit numbers, 32-bit numbers, etc. Although the language tries to hide this overloading, its effects are easily felt at the size boundaries of the types. It is a common source of confusion among programmers, and so a common source of bugs. We give a few examples that reveal that arithmetic in C is heavily overloaded, and that even trivial programs can involve complex semantics. Keep in mind that unless specified, in C a type is assumed to be signed.[4]

For the purposes of these examples, assume that **int**s are 2 bytes (capable of representing the values −32768 to 32767) and **long int**s are 4 bytes (−2147483648 to 2147483647). In the following program, what value does *c* receive [39, Q3.14]?

int a = 1000, b = 1000;
long int c = a ∗ b;

One is tempted to say 1000000, but that misses an important C-specific detail. The two operands of the multiplication are **int**s, so the multiplication is done at the **int** level. It therefore overflows ($1000 * 1000 = 1000000 > 32767$), which, according to the C standard, makes the expression undefined.

Let us change the example slightly by making the types of *a* and *b* unsigned (0 to 65535):

unsigned int a = 1000, b = 1000;
long int c = a ∗ b;

Here, the arithmetic is again performed at the level of the operands, but overflow on *unsigned* types is completely defined in C. It yields a value by "repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type" [21, §6.3.1.3:2]. So, we can take our mathematical 1000000 and continually subtract 65536 from it until the result in the correct range: $1000000 - 15 * 65536 = 16960$. For unsigned types, this is essentially computing the modulus.

One last variation—**signed char**s are one byte in C (−128 to 127).[5] What does *c* receive?

signed char a = 100, b = 100;
int c = a ∗ b;

Since the chars are signed, then based on the first example above the result would seem undefined ($100 * 100 = 10000 > 127$). However, this is not the case. In C, types smaller than **int**s are promoted to **int**s before doing arithmetic. There are basically implicit casts on the two operands: **int** c = (**int**)a ∗ (**int**)b;. Thus, the result is actually 10000.

While the above examples might seem like a game, the conclusion we draw is that it is critical when defining the semantics of C to handle *all* of the details. The semantics at the higher level of functions and statements is actually much easier than at the level of expressions and arithmetic. These issues are subtle enough that they are very difficult to catch just by manually inspecting the code, and so need to be represented in the semantics if one wants to find bugs in real programs. This is one of our primary reasons for wanting an executable semantics.

[4] Except **char**, where it is implementation-defined.

[5] We should note that bytes are only required to be at least 8 bits long. The particular numbers here are for the example only.

4. The Rewriting Semantics of C

In this section, we describe the different components of our definition and give a number of example rules from the semantics. We additionally describe the rewriting formalism used.

4.1 Rewriting Logic and $\mathbb{K}$

We use a rewriting-based semantic framework called the $\mathbb{K}$ formalism [34], inspired by rewriting logic (RL) [25]. In particular, our machine-readable semantics is written using the K-Maude tool [37], which takes $\mathbb{K}$ rewrite rules and translates them into Maude [5]. Maude is a performant rewriting-logic engine that provides facilities for the execution and/or analysis of rewriting-logic theories. This enables us to use the definition as an interpreter (Section 4.7), as well as allows us to demonstrate its suitability for program analysis (Section 6). For a complete introduction to this formalism, see Appendix A.

RL organizes term rewriting *modulo equations* (namely associativity, commutivity, and identity) as a logic with a complete proof system and initial model semantics. The central idea behind using RL as a formalism for the semantics of programming languages is that the evolution of a program can be described using rewrite rules. A rewriting theory consists essentially of a signature describing terms and a set of rewrite rules that describe the actual steps of computation. Given some term allowed by signature (in most cases, a program together with input), deduction consists of the application of the rules to that term. This yields a transition system for any program, and there are pre-existing generic tools that allow different means to explore the transition system. A single path of rewrites describes the behavior of an interpreter, while searching all paths would yield all possible answers in a non-deterministic program. These techniques are explored in Section 6.

For the purposes of this paper, the $\mathbb{K}$ formalism can be regarded as a front-end to RL designed specifically for defining languages. In $\mathbb{K}$, parts of the state are represented as nested multisets, as seen in Figure 2 and described in Section 4.3. These collections contain pieces of the program state like a computation stack or continuation (e.g., k), environments (e.g., env, types), stacks (e.g., callStack), etc. As this is all best understood through an example, let us consider a rule of C:

$$\langle \frac{\&X}{\text{tv}(Loc, \text{ptrType}(T))} \cdots \rangle_{\text{k}} \langle \cdots X \mapsto Loc \cdots \rangle_{\text{env}} \langle \cdots X \mapsto T \cdots \rangle_{\text{types}}$$

First, recall that this is describing a rewrite-rule, not a natural deduction system. It says that if the next thing to be evaluated (also called a redex) is the application of the referencing operator (&) to a variable X, then one should look up X in both the environment and the map of types to find its location Loc in memory and type T, respectively. With this information, one should transform the redex into a typed-value pair tv(Loc, ptrType(T)). That is to say, the result of applying the "&" operator to variable X of type T is a value of type pointer-to-T which is the location of X.

Although there is a lot going on here, this example exhibits a number of features of our chosen formalism. First, rules only need to mention those cells (again, see Figure 2) relevent to the rule. Second, to omit a part of a cell we write "···". For example, in the above k cell, we are only interested in the current redex $\&X$, but not the rest of the context. Finally, we draw a line underneath parts of the state that we wish to change—in the above case, we only want to evaluate part of the computation, but neither the context nor the environment or type maps change.

Again, since this formalism is relatively unknown, we must spend a little time describing why this unconventional notation is actually quite useful. The above rule would be written out as a

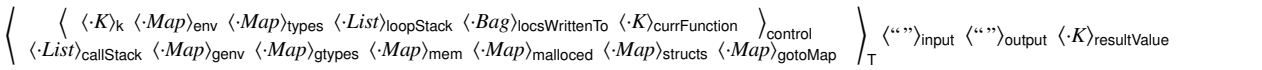


Figure 2. Subset of the C Configuration

traditional rewrite rule like this:

$$\langle \&X \curvearrowright K\rangle_{\mathsf{k}}\ \langle Env\ X \mapsto Loc\rangle_{\mathsf{env}}\ \langle TEnv\ X \mapsto T\rangle_{\mathsf{types}}$$
$$\Rightarrow \langle \mathrm{tv}(Loc, \mathrm{ptrType}(T)) \curvearrowright K\rangle_{\mathsf{k}}\ \langle Env\ X \mapsto Loc\rangle_{\mathsf{env}}\ \langle TEnv\ X \mapsto T\rangle_{\mathsf{types}}$$

This rule says the same thing as the original rule, but nearly the entire rule is duplicated on the right-hand side (RHS).

4.2 Syntax

We use CIL [26], an "off-the-shelf" C parser and transformation tool, to parse and simplify the original C syntax. After parsing, a custom pretty-printer then writes the abstract syntax tree (AST) back out in a form parsable by Maude. While CIL is an excellent parser, some of its transformations cannot be disabled, which is disadvantageous to us (see Section 5), although useful to many others. Blazy and Leroy [2], who also use CIL as the front-end to their semantics, chose to modify CIL's source code to prevent some transformations from taking place. In order to take advantage of any new versions that might be introduced, we chose not to modify CIL.

In some rare instances we were able to find bugs in CIL. These bugs were related to the interpretation of literals and the particular type promotions applied to function arguments when prototype information was not available. In some cases, the bug was simply the omission of a necessary cast. We believe these have not been detected by other users because the resulting code is not actually wrong, it just did not match CIL's claims (i.e., that all implicit casts would be made explicit). Because of this, compilers operating on the output of CIL would insert their own casts and no bug would be detected. While we reported the bugs (two have been fixed, but there is still one outstanding), in the end we also added the semantics of these features to our definition so that we handle all implicit type conversions.

4.3 Configuration

The configuration of a running program is represented by nested multisets of labeled cells, and Figure 2 shows the most important cells used in our semantics. The large T cell contains the cells used during program evaluation: at the top, a control cell containing cells related to local control flow, and below, a number of cells dealing with global information.

In the control cell, there is a k cell containing the current computation itself, a local variable environment (env), a local type environment (types), a loop stack (Section 4.5.2), a record of the locations that have been written to since the last sequence point (Section 4.6), and the name of the current function. The cells inside the control cell were separated in this manner because these are the cells that get pushed onto the call stack when making a function call. Outside the control cell are a number of global mappings, such as the call stack, the global variable environment (genv), the global type environment (gtypes), the heap (mem), the dynamic allocation map (malloced), aggregate definitions (structs), and a map from function-name/label pairs to continuations (for use by `goto` and `switch`). Finally, outside the T cell, there are cells for input, output, and a final cell for the value returned by `main()`.

As mentioned above, we keep maps for the types of identifiers. One could annotate variables and expressions with their types at parse time (this is what Blazy and Leroy [2] do), but we are not currently doing this. Annotations would make executing the semantics more efficient, but it is otherwise equivalent.

4.4 Memory Model

Our memory is essentially a map from locations to blocks of bytes. It is based on the memory model of both Blazy and Leroy [2] and Roşu et al. [35] in the sense that the actual locations themselves are symbolic numbers. However, it is more like the former in that the actual blocks of bytes are really maps from offsets to bytes. When new objects (arrays, structs, etc.) get allocated, each is created as a new block and is mapped from a new symbolic number. The block is allowed to contain as many bytes as in the object, and accesses relative to that object must be contained in the block. We represent information smaller than the byte (i.e., bitfields) by using offsets within the bytes themselves. While it would make things more consistent to treat memory as mappings from bit locations to individual bits, bitfields themselves are not addressable in C, so we decided on this hybrid approach.

4.5 Semantics

We now give the flavor of our semantics through a few of the simpler rules. In the interests of space, we can only cover a small part of the language. The full semantics can be found in Appendix B.

4.5.1 Lookup and Assignment

First, all assignments are converted to a helper operator called "assign", underneath which the left-hand sides (LHSs) are referenced, then dereferenced. This is valid since the left operand of an assignment operator is a modifiable lvalue [21, §6.5.16:2].

$$\langle \frac{E_1 = E_2}{\mathrm{assign}(*(\&E_1), E_2)} \cdots \rangle_{\mathsf{k}}$$

We do this in order to give the semantics for "x = 5" the same way as "*p = 10", as the first becomes "$\mathrm{assign}(*(\&(x)), 5)$", and the second "$\mathrm{assign}(*(\&(*(p))), 10) \equiv \mathrm{assign}(*(p), 10)$", at which point both have a dereference at the top of the LHS. The assign operator is given strictness attributes (evaluation strategies) to cause the appropriate positions to be evaluated: "CONTEXT: $\mathrm{assign}(\text{—}, \square)$" and "CONTEXT: $\mathrm{assign}(*(\square), \text{—})$". These rules say that, first, the RHS of an assignment is to be evaluated before the assignment can be evaluated, and second, that any term inside a dereference operator at the top of the LHS is to be evaluated before the assignment can be evaluated. No restrictions are placed on the order or allowed interleavings of these evaluations.

We rely on these strictnesses to take care of transforming those positions into values. We can then give the semantics of assignment for when those positions are values:

$$\langle \frac{\mathrm{assign}(*\mathrm{tv}(Loc, \mathrm{ptrType}(T)),\ \mathrm{tv}(V,\ T)) \cdots}{\mathrm{putInMem}(Loc,\ \mathrm{tv}(V, T)) \curvearrowright \mathrm{tv}(V,\ T)} \rangle_{\mathsf{k}}$$

Recall the "tv" construct from Section 4.1: it is how values are represented internally in our semantics (tv stands for "typed value"), where the first argument is the raw data and the second argument is the type. In C, all values are typed. Numerical literals must be assigned a type in order to determine how they will behave (as seen in Section 3), and blocks of memory must be accessed through variables of a particular type. With that in mind, this rule says that once operands have evaluated properly, the RHS value can be stored in memory, and the assignment itself evaluates to that value. It also ensures the types correspond properly.

Now we look at getting values back from memory. We will focus on lookup done through a simple identifier. This is conceptually

simpler than assignment, though there are three cases. The first is a simple variable with some simple type:

$$\langle \frac{X}{\text{readMem}(Loc, T)} \cdots \rangle_{\text{k}} \langle \cdots X \mapsto Loc \cdots \rangle_{\text{env}} \langle \cdots X \mapsto T \cdots \rangle_{\text{types}} \quad \text{when } \neg\text{isArrayType}(T) \wedge \neg\text{isFunctionType}(T)$$

We must look up, in the environment, the location of the variable in memory. We also need to know the variable's type (to eventually figure out its representation in memory), so we look it up in a map of the declared types of variables. With this information, we then reduce the variable lookup to the helper operator "readMem", which deals with the actual details of gathering bytes from memory.

The other two cases are actually simpler. When the identifier is of array type, it evaluates to its location [21, §6.3.2.1:3]:

$$\langle \frac{X}{\text{tv}(Loc, \text{ptrType}(T))} \cdots \rangle_{\text{k}} \langle \cdots X \mapsto Loc \cdots \rangle_{\text{env}} \langle \cdots X \mapsto \text{arrType}(T, \_) \cdots \rangle_{\text{types}}$$

Despite the common knowledge that "arrays are pointers", this is actually far from the truth. In C, arrays are second-class objects—they have no value by themselves and so are evaluated to their location.[6] This is different from the way pointers (and other first-class objects) evaluate. This can be seen clearly by examining the two rules above. The first lookup rule applies to pointer types, while the second rule handles array types. The former performs a lookup in the environment, and then requests a read from memory. The latter uses the environment only, at which point it is finished evaluating. No read from memory is performed. It is because of this difference that arrays cannot be assigned, passed as values, or returned.

Lastly, when the identifier is of function type, it also evaluates to its location [21, §6.3.2.1:4]:

$$\langle \frac{X}{\text{tv}(Loc, \text{ptrType}(T))} \cdots \rangle_{\text{k}} \langle \cdots X \mapsto Loc \cdots \rangle_{\text{env}} \langle \cdots X \mapsto T \cdots \rangle_{\text{types}} \quad \text{when isFunctionType}(T)$$

but the resulting type is different than with arrays.

4.5.2 While and Break

In comparison to expressions, the semantics of C statements is simple (all together it takes up fewer than 10 pages in the C1X standard [21, §6.8]). Still, we now show our semantics for `while` and `break`. The first rule prepares a `while` loop for execution:

$$\langle \frac{\text{while}(B)S \curvearrowright K}{\text{preparedWhile}(B)S \curvearrowright \text{break}} \rangle_{\text{k}} \langle \frac{\cdot}{K} \cdots \rangle_{\text{loopStack}}$$

The remaining computation (K) following the `while` (i.e., the context of the `while`) is pushed onto a loop stack. The effect is a push, because "nothing" (represented by "$\cdot$") at the top of the stack is being replaced by K. This loop stack will be used if the loop terminates normally or a `break` statement is encountered. We insert a break statement into the computation after the `while` to use a single mechanism in either case.

The main rule for `while` loops simply unrolls the (marked) loop once, turning the guard into an `if`-statement:

$$\langle \frac{\text{preparedWhile}(B)S}{\text{if}(B)(S \curvearrowright \text{preparedWhile}(B)S)} \cdots \rangle_{\text{k}}$$

When the guard evaluates as true, the body and then the prepared `while` will be up for evaluation again. Using the loop stack, the `break` statement itself is trivial:

$$\langle \frac{\text{break} \curvearrowright \_}{K} \rangle_{\text{k}} \langle \frac{K}{\cdot} \cdots \rangle_{\text{loopStack}}$$

It simply pops the stack and replaces the current computation with what was popped.

[6] An exception to this would be if an array is inside of a `sizeof` operator. We handle this by virtue of `sizeof` not being strict.

4.5.3 Goto

Finally, here is the rule for `goto`:

$$\langle \frac{\text{goto}(X) \curvearrowright \_}{K} \rangle_{\text{k}} \langle F \rangle_{\text{currFunction}} \langle \frac{\_}{S} \rangle_{\text{loopStack}} \langle \cdots (F, X) \mapsto (K, S) \cdots \rangle_{\text{gotoMap}}$$

This rule describes the high-level behavior of a `goto` statement. It says that when the top item in the computation is a `goto` statement to label X, then match the current function name F from the currFunction cell, and use it together with the label to look up in the gotoMap cell. This cell is a mapping from pairs (function names and label names) to pairs (computations and loop stacks). Finally, replace the current computation and loop stack with the results of the lookup.

It is worth mentioning that the semantics of C allows `goto`s to proceed over variable declarations or change scopes entirely, unlike the semantics of C++. Because CIL hoists variable declarations to the tops of functions, this difficulty is hidden from us.

4.5.4 Parametric Behavior

We chose to make our definition parametric in the implementation-defined behaviors (and are not the first to do so [2, 7]). Thus, one can configure the definition based on the architecture or compiler one is interested in using, and then proceed to use the formalism to explore behaviors. This parameterization allows the definition to be "fleshed out" and made executable.

For a simple example of how the definition is parametric, our K-Maude module `C-SETTINGS` starts with:

numBytes(signed-char) = 1	numBytes(short-int) = 2
numBytes(int) = 4	numBytes(long-int) = 4
numBytes(long-long-int) = 8	numBytes(float) = 4
numBytes(double) = 8	numBytes(long-double) = 16

These settings are then used to define a number of derived operators:

$$\begin{aligned} \text{numBits}(T) &= \text{numBytes(T)} * \text{bitsPerByte if } \neg\text{hasBitfieldType}(T) \\ \text{min(int)} &= -(2^{\text{numBits(int)}-1}) \\ \text{max(int)} &= 2^{\text{numBits(int)}-1} - 1 \end{aligned}$$

and finally, a further derived equation defining how an integer V of type T is cast to an unsigned integer type T':

$$\begin{aligned} &\text{cast}(T', \text{tv}(V, T)) = \text{tv}(T', V\%(\max(T') + 1)) \\ &\text{if isIntegerType}(T) \wedge \text{isUnsignedIntegerType}(T') \wedge V > \max(T') \end{aligned}$$

There are many similar equations to define other cases, but the above is only meant to convey the idea.

In principle, there are many things that could be made customizable like this, including how arithmetic is performed (one's or two's complement or signed magnitude), how types are aligned, or the way fields, including bitfields, are packed. In our current semantics, only some of these are immediately changeable.

4.6 Expression Evaluation Strategy

The standard allows compilers as much freedom as possible in optimizations, which includes allowing them to choose their own expression evaluation order. This includes allowing them to:

- delay side effects: e.g., allowing the write to memory required by x=5 or x++ to be made separately from its evaluation or use;
- interleave evaluation: e.g., A + (B + C) can be evaluated in the order B, A, C.

At the same time, the programmer must be able to write programs whose behaviors are reproducible, and only allow non-determinism in a controlled way. Therefore, the standard also makes undefined certain situations where reordering creates a "race condition". The latest treatment of this restriction is given by the C1X standard:

> If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings [...], the behavior is undefined if such an unsequenced side effect occurs in any of the orderings [21, §6.5:2].

This essentially means that without crossing any sequence points, any memory location can only be written to once, and never read after being written to. Furthermore, it is the case that if there exists *any* possible execution that violates the above restriction, then the expression is to be considered undefined. We explore a number of examples of this in Section 6.3.1.

"Sequenced before" is a relation defined in places throughout the standard to make explicit the orderings that need to be considered. One of the most important parts is that for any expression, the evaluation of operands is sequenced before the evaluation of the operator itself. This relation is related to the concept of "sequence points", also defined by the standard. All previous evaluations and side effects must be complete before crossing sequence points.

Our semantics does not describe the reordering of side-effects, because it turns out that such (rather expensive) reorderings are unnecessary for the purposes of catching undefined expressions or enumerating the behaviors of defined ones. What would it mean for there to exist an expression whose definedness relied on whether or not a side effect (a write) occurs later instead of earlier? There must be three parts to the expression: a subexpression E generating a side effect X, and, for generality's sake, further subexpressions E' and E''. The particular evaluation where we do side effects immediately would look like $E\ X\ E'\ E''$. Because this is always a possible execution, and we assume it does not show a problem, we can conclude neither E' nor E'' may neither read or write to X. If there is a problem only when we delay the side effect, it can be seen in a path like $E\ E'\ X\ E''$. For this to be different than applying the changes to X immediately, it means there must be some use of X in the evaluation of E'. But this contradicts the previous assumption.

Therefore, for the purposes of determining if an expression is undefined, as long as we are doing all possible interleavings of evaluation and keeping track of reads and side effects, it is sufficient to do the write immediately upon evaluation. Our semantics does capture this behavior as seen in Section 6.3.1.

4.7 KCC

Using some simple shell and Perl scripts for handling output and input, C programs are parsed with CIL and translated into a Maude term, then reduced using the rules of our formal semantics, producing indistinguishable behavior from the same C program run as native code. We call this "compiler" KCC. Once KCC is installed on a system, compilation of C programs generates a single executable file (an "a.out") containing the semantics of C, together with a parsed representation of the program and a call to Maude. The output is captured by a script and presented so that for working programs the output and behavior is identical to that of a real C compiler. To emphasize the seamlessness, here is a simple transcript:

```
$ kcc helloworld.c
$ ./a.out
Hello world
```

While it may seem like a gimmick, it helped our testing and debugging tremendously. For example, we could run the definition using the same test harness GCC uses for its testing (see Section 7).

5. Limitations

Here we delineate the limitations of our definition and explain their causes and effects.

To use K-Maude to define our semantics, we had to rely on the parsing capabilities of Maude itself when reading in programs. The capabilities of Maude's parser reflect the syntactic constraints under which it operates—one can think of the Maude language itself as having a user-definable syntax, which makes generic parsing difficult. Because of this, we turned to CIL for frontend parsing.

CIL made getting started on the semantics much easier than it otherwise would have been, but came at a price. In order to keep their generated AST simple, CIL has a number of builtin simplifications that are impossible to disable. However, this means that some constructs never appear in the output of CIL, such as `for` or `do-while` loops, static variables, the ternary expression (`_?_:_`), boolean operators, and nested scope. CIL also has limited or lacking support for variable length arrays and **_Complex** types. Because of this, we do not give semantics for these constructs. Also, CIL rewrites complex expressions into a series of statements using temporary variables. This is a great inconvenience for us because it requires us to restore the AST in order to explore different execution strategies (as demonstrated in Section 6.3.1).

The behaviors we are describing as limiting were choices purposefully made by the designers of CIL. In some sense, CIL is intended to be a structured subset of C to help make analyses simple. Their tool is good at the jobs for which it is intended. However, we believe that for us, the only way to ensure we cover the semantics of the entire language is by switching to using a C parser only, which is future work.

We have not yet handled **_Bool** types or wide characters, as they did not show up in a significant number of the programs we looked at so far. We also do not handle any parts of the specification related to alignment. We have no alignment restrictions, or equivilently, we align all types to one-byte boundaries. This means we cannot catch undefined behavior related to alignment restrictions. We should note that others have worked on formalizing alignment requirements [27], but it has never been incorporated into a full semantics for C.

We have not yet used our C definition for doing language or program level proofs, even though the $\mathbb{K}$ Framework supports both program level [36] and semantics level proofs [11]. To do so, we need to extend our semantics with support for formal annotations (e.g., assume, assert, invariant) and connect it to a theorem prover. In future work, we intend to apply those techniques to C.

Finally, we support a relatively small subset of the standard library, which has made it difficult to run many programs "in the wild". We also intend on addressing this, at the least by obtaining available implementations written in C.

6. Applications

Here we describe applications of our formal semantics, other than the interpreter already described.

6.1 Debugging

By introducing a special function "`debug()`" that acts as a breakpoint, we can turn the Maude debugger into a simple debugger for C programs. In the semantics, we handle this function specially by giving a labeled rule that causes it to evaluate to a "void" value. It is essentially a no-op, but one on which we can instruct Maude to break. If this function is called during execution, it drops the user into a debugger that allows her to inspect the current state of the program. She can step through more rules individually from there, or simply note the information and proceed. If the `debug()` call is inside a loop, the user will see a snapshot each time it reaches the expression. This debugging can be turned on or off at program invocation.

6.2 Runtime Verification

There are two main avenues through which we can catch and identify runtime problems with a program: catching undefined behavior, and symbolic execution.

6.2.1 "Core Dumping"

The first mechanism is based around the idea that when something lacks semantics (i.e., when its behavior is undefined according to the standard) then the evaluation of the program will simply stop when it reaches that point in the program. We use this mechanism to catch errors like signed overflow or array out-of-bounds.

In this small piece of code, the programmer forgot to leave space for a string terminator ('\0'). The call to `strcpy()` will read off the end of the array:

```
char dest[5], src[5] = "hello";
strcpy(dest, src);
```

GCC will happily execute this, and depending on the state of memory, even do what one would expect. It is still incorrect, and our semantics will get stuck trying to read past the end of the array, and report the state of the configuration (a concrete instance of that shown in Figure 2) and what exactly it was trying to do next.

6.2.2 Symbolic Execution

Through the use of symbolic execution, we can further enhance the above idea by expanding the behaviors that we consider undefined, while maintaining the good behaviors. For example, we can treat pointers not as concrete integers, but as symbolic values. These values can then have certain behavior defined on them, such as comparison, difference, etc. This technique is based on the idea of *strong memory safety*, which had previously been explored with a simple C-like language [35]. In this context, it takes advantage of the fact that addresses of local variables and memory returned from allocation functions like `malloc()` are unspecified [21, §7.20.3]. However, there are a number of restrictions on many addresses, such as the elements of an array being completely contiguous and the fields in a struct being ordered (though not necessarily contiguous).

For example, take the following small fragment of code:

```
int a, b;
if (&a < &b) { ... }
```

The behavior is conditioned on the particular allocation strategy of an implementation to determine what code to execute. While it could be argued that this should be allowed (it relies on unspecified behavior, not undefined), we choose not to implement unspecified behavior whenever possible. In our semantics, evaluation gets stuck trying to determine if the two symbolic locations are comparable. However, as in the case of the specification, sometimes locations are comparable. If we take the following code instead:

```
struct { int a; int b; } s;
if (&s.a < &s.b) { ... }
```

the addresses of a and b are guaranteed to be in order [21, §6.2.5:20], and in fact runs fine in our semantics.

Although we try to only give implementation-defined behavior, in some cases we also give unspecified behavior. One such example is a partially initialized struct. In order to copy a struct one byte at a time (as in an implementation of `memcpy()`), every byte needs to be copied. Uninitialized fields (or padding) should still be copied, and no error should occur [21, §6.2.6.1:5–7]. Using concrete values here would mean missing some incorrect programs, so we use symbolic values that allow reading and copying to take place as long as the program never uses those uninitialized values.

6.3 State Space Search

Again taking advantage of the tools provided by Maude, we can do both matching-based state search or explicit state model-checking with linear temporal logic. We show examples of the former to find undefined behavior, but due to space concerns, only here mention the latter as possible with this technique [12]. While the examples below are basic, they show how our semantics captures the appropriate expression evaluation semantics precisely.

6.3.1 Exploring Evaluation Order

To show our semantics captures the evaluation orders of C expressions allowed by the specification, we show here some examples from related works. When we say that our semantics gives a set of results, we mean that we actually performed the search using the tools provided by Maude. For these examples, note that the comma operator provides a sequence point [21, §6.5.17:2].

To start with a simple example from Papaspyrou and Maćoš [33], we take a look at x=0,x+(x=1). This expression is undefined because it is possible that the right sub-expression of the addition (the assignment) is evaluated before the left sub-expression (the lone x). Using our semantics to do a search of the behaviors of this expression yields two possible behaviors: `{Error}`[7] and `{x=1,e=1}`, where `e` is the result of the entire expression.

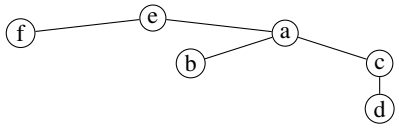
Norrish [29] offers the deceptively simple addition expression (x=0) + (x=0), which in many languages would be valid. However, in C it is again a technically undefined expression due to the multiple assignments to x with no intervening sequence point. Our semantics gives two `{Error}` states, one for each path. All paths are undefined, so the naively-expected result `{x=0,e=0}` is not returned.

Another example in the literature is given by Papaspyrou [32], which shows how C can exhibit non-deterministic behavior while staying conformant. The driving expression is the addition of two function calls. In C, function evaluation is not allowed to interleave [21, 6.5.2.2:10], so the behavior of this program is determined solely on which call happens last:

```
int r = 0;
int f (int x) { return (r = x); }
int main(void){ return (f(1) + f(2), r); }
```

If `f()` is called with the argument 2 last then the result will be 2, and vice-versa. Searching with our semantics gives the behaviors `{r=1}` and `{r=2}`, which are indeed the two possible results.

As a last example, we look at a more complex expression of our own devising: f()(a(b(), c(d()))). Except for `f()`, each function call simply prints out its name and returns 0. The function `f()`, however, prints out its name and then returns a function pointer to a function that prints `e`. The function represented by this function pointer will be passed results of `a()`. We elide the actual function bodies, because the behavior is more easily understood by this tree:



This tree (or Hasse diagram) describes the sequencing relation. That is, it must be the case that *d* happens before *c*, that *b* and *c* happen before *a*, and that *f* and *a* happen before *e*. Running this example through our search tool gives precisely the fifteen allowable behaviors.

[7] We take the liberty to replace the "stuck state" with "Error", as the actual state itself is many pages long.

7. Evaluation

No matter what the intended use is for a formal semantics, its use is limited if one can not generate confidence in its correctness. To this aim, we ensured that our formal semantics remained executable and computationally practical.

7.1 GCC Torture Tests

As discussed in Section 4.7, our semantics is encapsulated inside a drop-in replacement for GCC [15], which we call "KCC". This enables us to test the semantics as one would test a compiler. We were then able to run our semantics against the GCC C-torture-test [16] and compare its behavior to that of GCC itself, as well as the Intel C++ Compiler (ICC) and a CIL+GCC combination (i.e., running GCC on the output of CIL).

We use the torture test for GCC 4.4.2, specifically those tests inside the "testsuite/gcc.c-torture/execute" directory. We chose these tests because they focus particularly on portable (machine independent) executable tests. The README.gcc for the tests says, "The 'torture' tests are meant to be generic tests that can run on any target." We found that generally this is the case, although there are also tests that include GCC-specific features, which had to be excluded from our evaluation. There were originally 1057 tests, of which we excluded 299 tests because they used GCC specific extensions or builtins, they used the Complex or Boolean data types (which CIL does not support), they were machine dependent (based on the presence of a .x file), they used non-standard library functions, or because GCC, ICC, or CIL could not compile or parse the test. This left us with 758 tests. Further manual inspection revealed an additional 43 tests that were either non-conforming according to the standard, or that CIL failed to generate the correct parse tree, bringing us to a grand total of 715 viable tests.

In order to avoid "overfitting" our semantics to the tests, we randomly extracted about 200 tests from the initial 1057. Some of these were later disqualified, which left us with a 174 test subset, or about 25% of the conforming tests. We developed our semantics using only this subset (and other programs from Section 7.2). After we were comfortable with the quality of our semantics when running this subset, we ran the remaining tests. Out of 541 previously untested programs, we successfully ran 514 (95%). This methodology helps show that our semantics is not an enumeration of cases fit only for passing specific tests. Here is the full comparison:

	Unseen (541)		All (715)	
Compiler	Count	Percent	Count	Percent
GCC 4.1.2	537	99.3	710	99.3
ICC 11.1	540	99.8	714	99.9
CIL+GCC	536	99.1	709	99.2
KCC	514	95.0	686	95.9

The 715 tests represent about 21,000 SLOC, or 29 SLOC/file. Our semantics ran over 90% of these programs in under 5 seconds (each). An additional 6% completed in 10 minutes, 1% in 40 minutes, and 2% further in under 2 days. This leaves about 1% that never finished. While this is not terribly fast performance, especially when compared to compiled C, the reader should keep in mind that this is an interpreter obtained for free from a formal semantics. There were ten tests which we passed but at least one of the other compilers did not. We did not investigate these in detail, but they generally seem related to boundary conditions on literals.

Correctness Analysis While the real compilers did better than our semantics, we consider these results a success. They have had years to fix corner cases, and have been running against these tests under no 25% constraint. Additionally, our semantics is mechanized—we can add the missing parts of the language and compare again. Because the semantics is executable, we can incorporate all passing tests into a regression suite, so that adding features or fixing mistakes can only increase our accuracy. Finally, these failing tests reveal precisely what we are missing. Upon analysis of the failures, we discovered they could be categorized as follows:

Cause	Count	Percent
Timeout	9	31
Pointer Arithmetic	6	21
Misc. Missing Cases	5	17
Parsing	4	14
Bitfields	3	10
Variadics	2	7

Some of the timeouts are actually caused by Maude parsing the results of CIL's transformation. We have requested more information from the developers of Maude, but have not yet received any advice. The other timeouts were at runtime, and represented programs that are either computationally intensive or memory intensive.

The pointer arithmetic errors were intricately related to casting. While our semantics handles the case when a pointer with pointer type is added to an integer type, it does not handle the case where the pointer had first been cast to an integer type, as in the code "**int** a[2]; (**int**)&a[0] + 4;". This oversight on our part is correctable by adding additional cases, but we would like to handle the issue more generically. Although the behavior of pointer to integer conversion is implementation-defined, the standard states that, "The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment" [21, §6.3.2.3:3]. While a flat memory where pointers are concrete natural numbers would suffice, we would like to maintain a separation in order to catch more bugs. We believe a compromise can be found.

The parsing errors were simply rare cases that had not shown up before, and so we were not handling them in our CIL pretty-printer. For the bitfields, we were missing cases involving large bitfields that straddle bytes boundaries. Finally, the problems with variadics were due to the way we are handling the standard library component. Specifically, we were not allowing one to pass or copy `va_list` objects.

Coverage Analysis In order to have some measure of the effectiveness of our testing, we recorded the application of every semantic rule for 613 of the GCC torture tests we passed (we focused on the quick-running programs). The GCC torture tests do not focus on the standard library, so we excluded those rules from our analysis. We also excluded rules that were duplicating CIL's behavior.[8] These particular tests achieved semantic-rule coverage of 94% (309/329 rules).

In addition to getting a coverage measure, this process suggests another interesting application. For example, in the GCC tests looked at above, a rule that deals with casts to array types, e.g., "(**int**[5])p", was never applied. By looking at such rules, we can create new tests to trigger them. These tests would improve both confidence in the semantics as well as the test suite itself.

7.2 Exploratory Testing

We have also tested our semantics on programs gathered from around the web, including programs of our own design and from open source compilers. Not counting the GCC tests, we include nearly 10,000 SLOC in our regression tests which are run when making changes to the semantics. Each test is deterministic, and the output is compared automatically against the output of GCC. These tests include a number of programs from the LCC [18] and CompCert [2]

[8] We added a number of these rules to address bugs in CIL.

compilers. We also execute the "C Reference Manual" tests (also known as cq.c),[9] which go through Kernighan and Ritchie [24] and test each feature described in about 5,000 SLOC. When these tests are added to the GCC tests described above, it brings our rule-coverage to 97% (320/329 rules).

We can successfully execute Duff's Device [10], an unstructured `switch` statement where the `case`s are inside of a loop inside of the `switch` statement itself, as well as quines (programs whose output are precisely their source code), and a number of programs from the Obfuscated C Code Contest [28].

8. Conclusion

In this paper we showed how one can give a detailed definition of a real language using the rewriting semantics, and how that definition can be used to analyze the behaviors of particular programs.

In the future, we would like to make the definition fully parametric, in the sense that there should be a module where each item marked as "implementation-specific" in the standard should be represented in this module. This would increase our confidence that the core of the definition was flexible with respect to the decisions we took regarding the low-level details. We have also begun work on eliminating our reliance on CIL.

The entire definition of C is reproduced in Appendix B and the machine-readable code can be found at `https://code.google.com/p/k-framework/wiki/DefinitionOfC`.

References

[1] P. E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Brigham Young University, February 1998.

[2] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[3] M. Bofinger. *Reasoning about C programs*. PhD thesis, University of Queensland, February 1998.

[4] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, second edition, 1998.

[5] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[6] J. V. Cook and S. Subramanian. A formal semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, November 1994.

[7] J. V. Cook, E. L. Cohen, and T. S. Redmond. A formal denotational semantics for C. Technical Report 409D, Trusted Information Systems, September 1994.

[8] T. F. Şerbănuţă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois, 2010. URL `fsl2.cs.uiuc.edu/~tserban2/serbanuta-2010-thesis.pdf`.

[9] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[10] T. Duff. On Duff's device, 1988. URL `http://www.lysator.liu.se/c/duffs-device.html`. Msg. to the comp.lang.c Usenet group.

[11] C. Ellison, T. F. Şerbănuţă, and G. Roşu. A rewriting logic approach to type inference. In *19th International Workshop on Algebraic Development Techniques (WADT'08)*, volume 5486 of *LNCS*, pages 135–151, 2009.

[9] We have been unable to determine the author or origin of this test suite. Please contact us with any information.

[12] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In *16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501–505, 2004.

[13] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Ebberup, Denmark, Aug. 1986.

[14] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 2nd edition, 2001. ISBN 0-262-06217-8. URL `http://www.cs.indiana.edu/eopl/`.

[15] FSF. GNU compiler collection, 2010. URL `http://gcc.gnu.org`.

[16] FSF. C language testsuites: "C-torture" version 4.4.2, 2010. URL `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`.

[17] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, volume 702 of *LNCS*, pages 274–308, 1993.

[18] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[19] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:1999: Programming languages—C. Technical report, International Organization for Standardization, December 1999.

[20] ISO/IEC JTC 1, SC 22, WG 14. Rationale for international standard—programming languages—C. Technical Report 5.10, International Organization for Standardization, April 2003.

[21] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:201x: Programming languages—C. Committee draft, International Organization for Standardization, August 2008.

[22] D. M. Jones. *The New C Standard: An Economic and Cultural Commentary*. Self-published, December 2008. URL `http://www.knosof.co.uk/cbook/cbook.html`.

[23] K semantic framework website. K semantic framework website, 2010. URL `https://code.google.com/p/k-framework/`.

[24] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1978.

[25] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[26] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[27] M. Nita, D. Grossman, and C. Chambers. A theory of platform-dependent low-level software. In *35th ACM Symposium on Principles of Programming Languages (POPL'08)*, 2008.

[28] L. C. Noll, S. Cooper, P. Seebach, and L. A. Broukhis. The international obfuscated C code contest, 2010. URL `http://www.ioccc.org/`.

[29] M. Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, December 1998.

[30] M. Norrish. A formal semantics for C++. Technical report, NICTA, 2008. URL `http://nicta.com.au/people/norrishm/attachments/bibliographies_and_papers/C-TR.pdf`.

[31] N. S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, February 1998.

[32] N. S. Papaspyrou. Denotational semantics of ANSI C. *Computer Standards and Interfaces*, 23(3):169–185, 2001.

[33] N. S. Papaspyrou and D. Maćoš. A study of evaluation order semantics in expressions with side effects. *Journal of Functional Programming*, 10(3):227–244, 2000.

[34] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[35] G. Roşu, W. Schulte, and T. F. Şerbănuţă. Runtime verification of C memory safety. In *Runtime Verification (RV'09)*, volume 5779 of *LNCS*, pages 132–152, 2009.

[36] G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *13th International Conference on Algebraic Methodology and Software Technology (AMAST'10)*, LNCS, 2010. To appear.

[37] T. F. Şerbănuţă and G. Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In *8th International Workshop on Rewriting Logic and its Applications (WRLA'09)*, volume 6381 of *LNCS*, pages 104–122, 2010.

[38] S. Subramanian and J. V. Cook. Mechanical verification of C programs. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, January 1996.

[39] S. Summit. C programming FAQs: Frequently asked questions, 2005. URL `http://www.c-faq.com/`.

[40] W. Zimmermann and A. Dold. A framework for modeling the semantics of expression evaluation with abstract state machines. In *Abstract State Machines 2003: Advances in Theory and Applications*, volume 2589 of *LNCS*, pages 391–406, 2003.

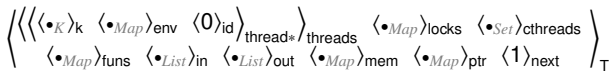


Figure 3. The initial configuration for executing KernelC programs.

A. Short Introduction to $\mathbb{K}$

This appendix gives a quick overview of the $\mathbb{K}$ framework—it is only an overview (and should be skimmed if you are comfortable with $\mathbb{K}$ or rewriting logic). This overview has been adapted (with permission) from Serbănută [8]. For a more extensive explanation of the $\mathbb{K}$ framework, see Rosu and Serbănută [34], Serbănută [8], and Serbănută and Rosu [37]. There is also a Google code project website [23] where the newest version of the K-Maude tool can be found.

In a nutshell, the $\mathbb{K}$ frameworks relies on computations, configurations, and $\mathbb{K}$ rules in giving semantics to programming language constructs. *Computations*, which gave the name of the framework, are lists of tasks, including syntax and have the role of capturing the sequential fragment of programming languages. *Configurations* of running programs are represented in $\mathbb{K}$ as bags of nested cells, with a great potential for concurrency and modularity. $\mathbb{K}$ *rules* distinguish themselves by specifying only what is needed from a configuration, and by clearly identifying what changes, and thus, being more concise, more modular, and more concurrent.

To exemplify the $\mathbb{K}$ framework, in this section we will use here a modified subset of C called KernelC (the full C semantics can be found at the end of this document). KernelC defines a subset of the C language which is nevertheless non-trivial, containing functions, memory allocation, pointers, and pointer arithmetic, and input/output primitives. It is expressive enough to be able to write C functions as the next one (which can be used for copying zero-terminated arrays):

```
void arrCpy(int * a, int * b) {
 while (* a ++ = * b ++) {}
}
```

Configurations. The initial running configuration of KernelC is presented in Figure 3. The configuration is a nested multiset of labeled cells, in which each cell can contain either a list, a set, a bag, or a computation. The initial KernelC configuration consists of a top cell, labeled "T" holding a bag of cells, among which a map cell, labeled by "mem", to map locations to values, a list cell, labeled "in", to hold input values and a bag cell, labeled "threads", which can hold any number of "thread" cells (signaled by the star $*$ attached to the name of the cell). The thread cell is itself a bag of cells, among which the "k" cell which holds a computation, which plays the role of directing the execution.

Syntax and Computations. Computations extend syntax with a task sequentialization operation, "$\curvearrowright$". The basic unit of computation is a task, which can either be a fragment of syntax, maybe with holes in it, or a semantic task, such as the recovery of an environment. Most of the manipulation of the computation is abstracted away from the language designer via intuitive PL syntax annotations like strictness constraints which, when declaring the syntax of a construct also specify the order of evaluation for its arguments. Similar decompositions of computations happen in abstract machines by means of stacks [13, 14], and also in the refocusing techniques for implementing reduction semantics with evaluation contexts [9]. However, what is different here is that $\mathbb{K}$ achieves the same thing *formally*, by means of rules (there are heating/slashcooling rules behind the strictness annotations, as explained below), not as an implementation means, which is what the others do.

The $\mathbb{K}$ BNF syntax specified below suffices to parse the program fragment `t = * x; * x = * y; * y = t;` specifying a sequence of statements for swapping the values at two locations in the memory syntax:

Exp ::= *Id*	
| * *Exp*	[strict]
| *Exp* = *Exp*	[strict(2)]
Stmt ::= *Exp* ;	[strict]
| *Stmt* *Stmt*	[seqstrict]

The strictness annotations add semantic information to the syntax by specifying the order of evaluation of arguments for each construct. The heating/cooling rules automatically generated for the strictness annotations above are:

$$\begin{array}{rcl} * \; ERed & \rightleftharpoons & ERed \curvearrowright * \; \square \\ E = ERed & \rightleftharpoons & ERed \curvearrowright E = \square \\ ERed \; ; & \rightleftharpoons & ERed \curvearrowright \square \; ; \\ SRed \;\; S & \rightleftharpoons & SRed \curvearrowright \square \;\; S \\ Val \;\; SRed & \rightleftharpoons & SRed \curvearrowright Val \;\; \square \end{array}$$

The heating/cooling rules specify that the arguments mentioned in the strictness constraint can be taken out for evaluation at any time and plug back in their original context. Note that the statement composition generates two such rules (as, by default, strictness applies to each argument); however, since the constraint specifies *sequential strictness*, the second statement can be evaluated only once the first statement was completely evaluated (specified by the *Val* variable which should match a value) and its side effects were propagated.

By successively applying these heating rules (from bottom towards top) on the statement sequence above we obtain the following computations:

$$\begin{array}{r}
\mathtt{t = *\,x;\ \ *\,x = *\,y;\ \ *\,y = t;} \quad \rightharpoonup \\
\mathtt{t = *\,x;} \curvearrowright \square\ \mathtt{*\,x = *\,y;\ \ *\,y = t;} \quad \rightharpoonup \\
\mathtt{t = *\,x} \curvearrowright \square\mathtt{;} \curvearrowright \square\ \mathtt{*\,x = *\,y;\ \ *\,y = t;} \quad \rightharpoonup \\
\mathtt{*\,x} \curvearrowright \mathtt{t} = \square \curvearrowright \square\mathtt{;} \curvearrowright \square\ \mathtt{*\,x = *\,y;\ \ *\,y = t;} \quad \rightharpoonup \\
\mathtt{x} \curvearrowright \mathtt{*}\square \curvearrowright \mathtt{t} = \square \curvearrowright \square\mathtt{;} \curvearrowright \square\ \mathtt{*\,x = *\,y;\ \ *\,y = t;} \quad\ \
\end{array}$$

To begin, because statement composition is declared sequentially strict, the left statement must be evaluated first. The strictness rule will pull the statement out for evaluation, and leave a hole in its place. Now an expression statement construct is at the top and, being strict, it requires that the assignment be puled out. Next, the assignment construct being strict in the second argument, its right hand side must be pulled out for evaluation. Finally, the dereferencing construct is strict, and the heating rule will pull out the identifier *x*. Thus, through the strictness rules, we have obtained the order of evaluation as a sequence of tasks.

$\mathbb{K}$ ***rules.*** Consider the following "swap" function for swapping the values at the locations pointed to by the arguments:

```
void swap(int * x, int * y){
 int t = * x;  * x = * y;  * y = t;
}
```

Assume we are in the middle of a call to "swap" with arguments "a" and "b" (which are mapped to memory locations 4 and 5, respectively), and assume that all statements but the last have already been executed, and that only the last statement is left to be executed and that *y* has already been evaluated to location 4. A running configuration corresponding to this situations could be the top configuration in Figure 4. By the strictness rules, we know that the next task to be executed is evaluating *t*.

Figure 4 shows how the $\mathbb{K}$ rule for reading the value of a local variable from the environment can be derived directly from the running configuration in which evaluating a local variable is the next task. First, through a process named *cell comprehension* we focus only on the parts of the cells which are relevant for this rule. At the same time, we can declare our intention to replace *t* by its value in the environment (which is 1) by underlining the part that needs to change and writing its replacement under the line, through what we call *in-place rewriting*. Finally, through the process of *configuration abstraction*, only the relevant cells are kept, and through *generalization* we replace the concrete instances of identifiers and constants with variables of corresponding types. The jagged edges are used to specify that there could be more content in the cell in addition to what is explicitly specified

The thus obtained $\mathbb{K}$ rule succinctly describes the intuitive semantics for reading a local variable: if a local variable *X* is the next thing to be evaluated and if *X* is mapped to a value *V* in the environment, then replace that occurence of *X* by *V*. Moreover, it does that by only specifying what is needed from the configuration, which is essential for obtaining modular definitions, and by precisely identifying what changes, which significantly enhances the concurrency.

Modularity. As mentioned above the configuration abstraction process is instrumental in achieving the desired modularity goal for the $\mathbb{K}$ framework. Relying on the initial configuration being specified by the designer, and the fact that usually the structure of such an configuration does not change during the execution of a program, the $\mathbb{K}$ rules are essentially invariant under change of structure. This effectively means that the same rule can be re-used in different definitions as long as the required cells are present, regardless or the additional context, which can be automatically inferred from the initial configuration. As an example, the $\mathbb{K}$ rule for reading the value of a local variable can be used for a configuration as the one specified above, for the full KernelC configuration, and even for a configuration in which the computation and the environment cells are in separate parts of the configuration like in the following case:

$$\left\langle \begin{array}{l} \langle\langle \mathtt{t} \curvearrowright *\ 5 = \square \curvearrowright \square\ ; \curvearrowright \mathtt{env}(\ a \mapsto 4\ \ b \mapsto 5\ )\rangle_{\mathsf{k}}\rangle_{\mathsf{thread}} \\ \quad \langle\langle \mathtt{x} \mapsto 4\ \ \mathtt{t} \mapsto 1\ \ \mathtt{y} \mapsto 5\rangle_{\mathsf{env}}\ \ \langle 4 \mapsto 7\ \ 5 \mapsto 7\rangle_{\mathsf{mem}}\rangle_{\mathsf{state}} \end{array} \right\rangle_{\top}$$

Power of expression. The structure of the computations, and the fact that the current task is always at the top of the computation as a similar effect on the power of expression, as configuration abstraction has for modularity. Let us give some examples of the easiness to define in $\mathbb{K}$ constructs which are known to be hard in other frameworks, thus arguing that the $\mathbb{K}$ framework reached the expressibility goal for an ideal definitional framework.

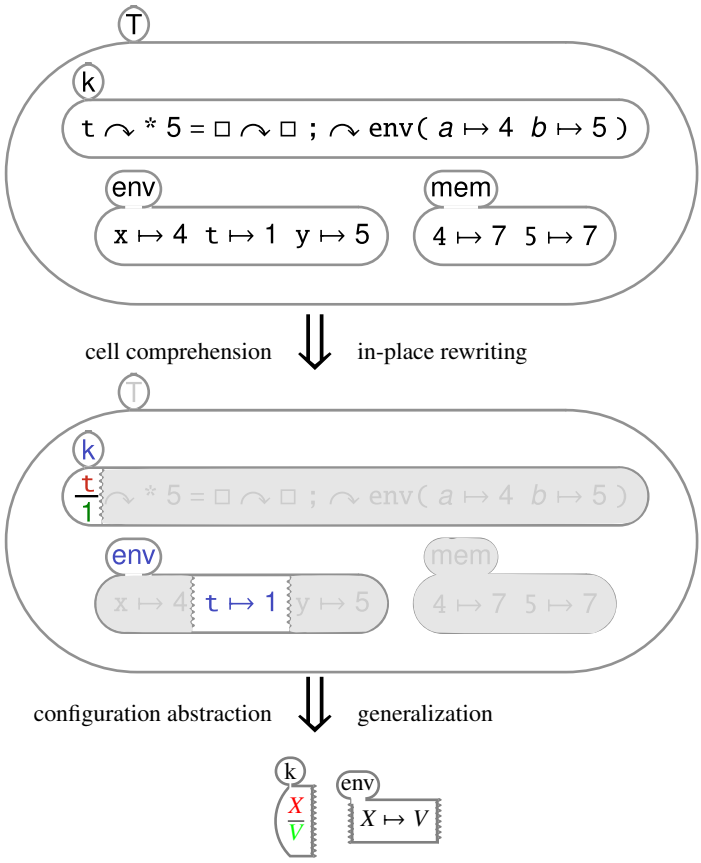


Figure 4. From running configurations to $\mathbb{K}$ rules

One first such example is the control intensive call with current continuation (call/cc), which is present in several functional languages (like Scheme), and to some extent, even in some imperative programming languages (such as the long-jump construct in C). Call/cc is known to be hard to capture in most frameworks (except reduction semantics with evaluation contexts) due to lack of access to execution context, which is there captured by the logical context, which lives at a meta-level which is not observable in the framework. Having the entire remainder of computation always following the current redex, allows $\mathbb{K}$ to capture this construct in a simple and succinct manner by the following two rules:

PASSING COMPUTATION AS VALUE: $\dfrac{\langle \texttt{callcc}\ V \curvearrowright K \rangle_{\text{k}}}{V \;\; cc(K)}$

APPLYING COMPUTATION AS FUNCTION: $\dfrac{\langle cc(K) \;\; V \curvearrowright \_ \rangle_{\text{k}}}{V \curvearrowright K}$

The first rule wraps the current reminder of the computation as a value and passes it to the argument of "`callcc`", which is supposed to evaluate to a function. If during the evaluation of that function call, the continuation value is applied to some value, then the reminder of the computation at that time is replaced by the saved computation to which the passed value is prefixed (as the result of the original `callcc` expression).

We cannot conclude the survey on the expressivity of the $\mathbb{K}$ framework without mentioning its reflective capabilities. Based on the fact that $\mathbb{K}$ regards all computational tasks as being abstract syntax trees, all language constructs become labels applied to other ASTs; for example the expression $a + 3$ is represented in $\mathbb{K}$ as $\_ + \_(a(\bullet_{List\{K\}}) \text{ , } 3(\bullet_{List\{K\}}))$. This abstract view of syntax allows reducing the computation-creating constructs to the following minimal core:

$$K ::= KLabel(List\{K\}) \mid \bullet_K \mid K \curvearrowright K$$
$$List\{K\} ::= K \mid \bullet_{List\{K\}} \mid List\{K\} \text{ , } List\{K\}$$

Moreover, this approach allows one to define a generic AST visitor, and in turn use that to define powerful reflective features such as code generation or a binder-independent substitution.

B. C Semantics

Here we present the full C semantics, generated directly from the original source code (which can be found on our Google project page at `https://code.google.com/p/k-framework/wiki/DefinitionOfC`). The LaTeX is being generated through a very new backend, and as such, there may be a number of typesetting/formatting or even correctness errors in the LaTeX. In particular, sometimes important parentheses have been discarded, which can change the meaning of side conditions. If there is any doubt, please consult the original source code at the above URL.

MODULE C-SEMANTICS
  IMPORTS C-SYNTAX
  IMPORTS C-CONFIGURATION
  IMPORTS COMMON-C-SEMANTICS
  IMPORTS DYNAMIC-C-SEMANTICS
END MODULE

MODULE COMMON-C-SYNTAX
  IMPORTS PL-BOOL
  IMPORTS PL-EXT-BOOL
  IMPORTS PL-NAT
  IMPORTS PL-INT
  IMPORTS PL-RAT
  IMPORTS PL-FLOAT
  IMPORTS PL-STRING
  IMPORTS PL-CONVERSION
  IMPORTS PL-QID
  IMPORTS PL-RANDOM
  IMPORTS PL-ID
  Program ::= *SeqList*
  Constant ::= *Integer-Constant* | *Floating-Constant*
  Integer-Constant ::= *Int*
                       | `U(` *Integer-Constant*)
                       | `L(` *Integer-Constant*)
                       | `UL(` *Integer-Constant*)
                       | `LL(` *Integer-Constant*)
                       | `ULL(` *Integer-Constant*)
                       | `hex(` *String*)
  Floating-Constant ::= *Float*
                        | `L(` *Floating-Constant*)
                        | `F(` *Floating-Constant*)
  String-Literal ::= *String*
  List{Expression} ::= *Expression*
                       | ·
                       | *List{Expression}* , *List{Expression}* [id: · strict hybrid assoc]
  Expression ::= *Id* | *Constant* | *String-Literal* | *Direct-Declarator*
                 | *Expression* [*Expression*]
                 | `Apply(` *Expression* , *List{Expression}*)
                 | `Apply(` *Expression*)
                 | *Expression* `.` *Id*
                 | *Expression* `->` *Id* [strict(1)]
                 | *Expression* `++`
                 | *Expression* `--` [strict(1)]
                 | `++` *Expression*
                 | `--` *Expression*
                 | `&` *Expression*
                 | `*` *Expression* [strict]
                 | `-` *Expression* [strict]
                 | `~` *Expression* [strict]
                 | `!` *Expression* [strict]
                 | `sizeof(` *Expression*)
                 | `Cast(` *Type-Name* , *Expression*) [strict]
                 | *Expression* `*` *Expression* [strict]
                 | *Expression* `/` *Expression* [strict]
                 | *Expression* `%` *Expression* [strict]
                 | *Expression* `+` *Expression* [strict]
                 | *Expression* `-` *Expression* [strict]
                 | *Expression* `«` *Expression* [strict]
                 | *Expression* `»` *Expression* [strict]
                 | *Expression* `<` *Expression* [strict]
                 | *Expression* `>` *Expression* [strict]

| *Expression* <= *Expression* [strict]
| *Expression* >= *Expression* [strict]
| *Expression* == *Expression* [strict]
| *Expression* != *Expression* [strict]
| *Expression* & *Expression* [strict]
| *Expression* ^ *Expression* [strict]
| *Expression* | *Expression* [strict]
| *Expression* *= *Expression*
| *Expression* /= *Expression*
| *Expression* %= *Expression*
| *Expression* += *Expression*
| *Expression* -= *Expression*
| *Expression* «= *Expression*
| *Expression* »= *Expression*
| *Expression* ^= *Expression*
| *Expression* |= *Expression*
| *Expression* &= *Expression*
| *Expression* = *Expression* [strict(2)]
| `Initializer`(*Expression*)
| `InitList`(*List{Expression}*)
| `InitItem`(*Expression*)
| `Designation`(*Expression* , *Expression*)
| `ArrayDesignator`(*Expression*)
| `FieldDesignator`(*Id*)

Init-Declarator-List ::= *Init-Declarator*

Init-Declarator ::= *Declarator*
| *Declaration* = *Expression* [strict(1)]
| *Declaration*={*List{Expression}* } [strict(1)]

Type-Specifier ::= *Typedef-Name* | *Struct-Or-Union-Specifier* | *Enum-Specifier* | *Base-Type*
| `Pointer`(*Type-Specifier*) [strict(1)]

Declarator ::= *Direct-Declarator*

Direct-Declarator ::= *Type-Specifier* | *Id*
| *Direct-Declarator* [*Expression*]
| `Pointer`(*Direct-Declarator*) [strict(1)]
| `Pointer`(*Direct-Declarator* , *Direct-Declarator*) [strict(1)]
| `Pointer`
| `Direct-Function-Declarator`(*Direct-Declarator* , *Parameter-Type-List*)
| `Direct-Function-Declarator`(*Parameter-Type-List*)
| `BitField`(*Expression*)
| `BitField`(*Declarator* , *Expression*)

List{Parameter} ::= *Struct-Declaration* | *Parameter-Declaration*
| ·
| *List{Parameter}* , *List{Parameter}* [id: · strict hybrid assoc]

Type-Name ::= *SeqList*

SeqList ::= *Type-Specifier* | *Storage-Class-Specifier* | *Type-Specifier* | *Declarator* | *Block-Item* | *External-Declaration*
| `Pointer`(*SeqList*)
| *SeqList* *SeqList*

Statement ::= *Labeled-Statement* | *Jump-Statement* | *Compound-Statement* | *Iteration-Statement* | *Expression-Statement* | *Selection-Statement*

Block-Item ::= *Init-Declarator* | *Statement* | *Declaration*

External-Declaration ::= *Function-Definition*
| `Global`(*SeqList*) [strict]
| Global()

Nat ::= loc(*Nat* , *Nat*)

Id ::= NULL

| calloc
| cos
| sin
| exit
| free
| main
| malloc
| printf
| putchar
| rand
| sqrt
| abort
| debug
| getchar
| exp
| log
| atan
| File-Scope
| fslOpenFile
| fslCloseFile
| fslFGetC
| fslPutc
| longjmp
| setjmp
| floor
| tan
| fmod
| atan2
| asin
| spawn
| sync
| lock
| unlock

Declaration ::= `Local`(*Declaration*) [strict]
| `Declaration`(*SeqList*) [strict]
| `Declaration`(*SeqList* , *Init-Declarator-List*) [strict(1)]
| `Typedef`(*SeqList* , *Declarator*) [strict(1)]

Storage-Class-Specifier ::= `typedef`
| `extern`
| `static`
| `auto`
| `register`

Typedef-Name ::= `typedefName`(*Id*)

Base-Type ::= void
| no-type
| float
| double
| long-double
| short-int
| unsigned-short-int
| char
| unsigned-char
| signed-char
| int

| unsigned-int
| short
| unsigned-short
| long-int
| unsigned-long-int
| long
| unsigned-long
| long-long
| unsigned-long-long
| long-long-int
| unsigned-long-long-int

Struct-Or-Union-Specifier ::= `structDef`(*Id* , *List{Parameter}*) [strict(2)]
| `unionDef`(*Id* , *List{Parameter}*) [strict(2)]
| `structDef`(*Id*)
| `unionDef`(*Id*)
| `struct`(*Id*)
| `union`(*Id*)

Struct-Declaration ::= `Field`(*SeqList* , *Declarator*) [strict(1)]

Enum-Specifier ::= `enum`(*Id* , *List{Expression}*)
| `enum`(*Id*)

Parameter-Type-List ::= `Parameter-Type-List`(*List{Parameter}*) [strict]
| Parameter-Type-List()

Parameter-Declaration ::= `...`
| `Parameter-Declaration`(*SeqList*) [strict]
| `Parameter-Declaration`(*SeqList* , *Declarator*) [strict(1)]

Labeled-Statement ::= *Id* : *SeqList*
| case(*Nat*) *Expression* : *SeqList* [strict(2)]
| default(*Nat*): *SeqList*

Compound-Statement ::= `Block`(*SeqList*)
| Block()

Expression-Statement ::= EmptyStatement;
| *Expression* ; [strict]

Selection-Statement ::= if(*Expression*) *Statement* [strict(1)]
| if(*Expression*) *Statement* else *Statement* [strict(1)]
| switch(*Nat*)(*Expression*) *Statement* [strict(2)]

Iteration-Statement ::= while(*Expression*) *Statement*

Jump-Statement ::= `goto` *Id*
| `break`
| `return`
| `return` *Expression* [strict]

Function-Definition ::= *Declaration*{*SeqList* } [strict(1)]

END MODULE

MODULE C-CONFIGURATION
IMPORTS C-SYNTAX
IMPORTS COMMON-C-CONFIGURATION
INITIAL CONFIGURATION:

$$\left\langle \left\langle \cdot Map \right\rangle_{\text{gotoMap}} \left\langle \cdot Map \right\rangle_{\text{genv}} \left\langle \cdot Map \right\rangle_{\text{mem}} \left\langle \text{loc}\,(\,1\,,\,0\,) \right\rangle_{\text{nextLoc}} \left\langle 0 \right\rangle_{\text{freshNat}} \left\langle 3 \right\rangle_{\text{nextFile}} \left\langle \cdot Map \right\rangle_{\text{malloced}} \left\langle \cdot Map \right\rangle_{\text{statics}} \left\langle \cdot Map \right\rangle_{\text{typedefs}} \left\langle \cdot Map \right\rangle_{\text{sizes}} \left\langle 0 \mapsto \text{“}\texttt{stdin}\text{”}\ 1 \mapsto \text{“}\texttt{stdout}\text{”}\ 2 \mapsto \text{“}\texttt{stdout}\text{”} \right\rangle_{\text{openFiles}} \right.$$
$$\left. \left\langle \left\langle \cdot Map \right\rangle_{\text{structs}} \left\langle \cdot Bag \right\rangle_{\text{busy}} \left\langle \cdot List \right\rangle_{\text{callStack}} \left\langle \left\langle \cdot List \right\rangle_{\text{buffer}} \left\langle \text{false} \right\rangle_{\text{blocked}} \left\langle \cdot K \right\rangle_{\text{k}} \left\langle \cdot Map \right\rangle_{\text{registers}} \left\langle \cdot Bag \right\rangle_{\text{locsWrittenTo}} \left\langle \text{File-Scope} \right\rangle_{\text{currentFunction}} \left\langle \cdot K \right\rangle_{\text{type}} \left\langle \cdot Map \right\rangle_{\text{env}} \left\langle \cdot List \right\rangle_{\text{loopStack}} \left\langle \cdot Bag \right\rangle_{\text{locals}} \left\langle \cdot Map \right\rangle_{\text{types}} \right\rangle_{\text{control}} \right\rangle_{\text{T}}$$
$$\left\langle \cdot Map \right\rangle_{\text{files}} \left\langle \text{“”} \right\rangle_{\text{input}} \left\langle \text{“”} \right\rangle_{\text{output}} \left\langle \cdot K \right\rangle_{\text{resultValue}}$$

END MODULE

```
MODULE INCOMING-MODULES
  IMPORTS K
  IMPORTS C-SYNTAX
  IMPORTS C-CONFIGURATION
  IMPORTS K-CONTEXTS
  IMPORTS K-PROPER
  IMPORTS K-QUOTED-LABELS
END MODULE
```

```
Module COMMON-SEMANTIC-SYNTAX
  imports INCOMING-MODULES
  KResult ::= Field | Value | Type
            | skipval
            | typedParameterList( List{K} ) [strict]
            | typedDeclaration( Type , K )
            | definedType( Type , K )
            | typedField( Type , Id )
  Expression ::= BaseValue | Value | SeqList
               | sizeof( Type )
               | HOLE
               | sizeofType( K ) [strict]
  State ::= String
  Type ::= Base-Type
         | arrayType( Type , Nat )
         | bitfieldType( Type , Nat )
         | functionType( Type , List{K} )
         | pointerType( Type )
         | structType( Id )
         | unionType( Id )
         | qualifiedType( Type , K )
  BaseValue ::= Nat | Int | Rat | Float | Bool
  ListItem ::= List( BagItem )
             | bwrite( Nat , BaseValue )
  Nat ::= piece( Nat , Nat )
        | trueUnknown
        | unknown( Nat )
        | bitloc( Nat , Nat , Nat )
        | inc( Nat )
        | charToAscii( String )
        | numBytes( Type ) [strat(1 0)memo]
        | numBits( Type ) [strat(1 0)memo]
        | numBitsPerByte
  Float ::= unknownF
  K ::= debugK
      | discard
      | memblock( Nat , Map )
      | l( KLabel )
      | bind( List{KResult} , List{KResult} )
      | declare( K , K ) [strict(1)]
      | converted( K , K )
      | evalToType
      | concretize( Value )
      | addTypes( K , K )
      | addGlobalTypes( K , K )
      | addGlobalTypes( K )
      | putInMem( Nat , K ) [strict(2)]
      | putInMem-aux( Nat , Value , Type , K ) [strict(4)]
      | putBytesInMem( Nat , List{K} , Type , K ) [strict(4)]
      | calcStructSize( List{KResult} )
      | calcUnionSize( List{KResult} )
      | calcStructSize-aux( List{KResult} , Rat )
      | calcUnionSize-aux( List{KResult} , Rat )
      | necessaryBytes( K ) [strict]
```

```
      | cast( Type , K ) [strict(2)]
      | arithInterpret( Type , BaseValue )
      | interpret( Type , BaseValue )
      | leftShiftInterpret( Type , BaseValue , K )
      | rightShiftInterpret( Type , BaseValue )
      | addLocals( Nat )
      | typeof( Expression )
      | writeToFD( Nat , Nat )
      | writeToFD( Nat , String )
      | readFromFD( Nat )
      | readFromFD( Nat , Nat )
      | figureOffset( Nat , K , Type ) [strict(2)]
      | calculateGotoMap( Id , K )
      | kpair( K , K )
      | kpair( K , List )
      | promote( K )
      | readFromMem( Nat , Type )
      | extractField( List{K} , Type , Id )
      | allocString( Nat , String )
      | popLoop
      | giveGlobalType( Type , K )
      | giveLocalType( Type , K )
      | application( K , List{K} ) [strict(1)]
      | case( K , BaseValue )
      | defaultCase( K )
      | sequencePoint
      | append( Nat , Nat , Value )
      | store Nat New K atLoc Nat [strict(2)]
      | storeNew K atLoc Nat [strict(1)]
      | store K atLoc Nat [strict(1)]
      | alloc( Nat , K ) [strict(2)]
      | allocWithDefault( Nat , K , Nat ) [strict(2)]
Id ::= unnamedBitField
     | anonymousId
Set ::= locations( List )
      | setOfTypes [strat(0)memo]
      | integerTypes [strat(0)memo]
      | unsignedIntegerTypes [strat(0)memo]
      | signedIntegerTypes [strat(0)memo]
Value ::= registerLocation( Id )
        | tv( BaseValue , Type )
        | atv( List{K} , Type )
        | Closure( K , K , K )
Bool ::= Set contains K [strat(1 2 0)memo]
       | isIntegerType( Type ) [strat(1 0)memo]
       | isUnsignedIntegerType( Type ) [strat(1 0)memo]
       | isSignedIntegerType( Type ) [strat(1 0)memo]
       | isFloatType( Type ) [strat(1 0)memo]
       | isArrayType( Type )
       | isFunctionType( Type )
Base-Type ::= enumType( Id )
            | registerInt
Char ::= firstChar( String )
String ::= butFirstChar( String )
```

List{K} ::= *Nat* to *Nat*
Int ::= min(*Type*) [strat(1 0)memo]
| max(*Type*) [strat(1 0)memo]
Map ::= undefRange(*Map* , *Nat* , *Nat*)
Bag ::= range(*Nat* , *Nat*)
Statement ::= loopMarked
MACRO: loc (*Block* , *N*) $<_{Int}$ loc (*Block* , *N′*) = *N* $<_{Int}$ *N′*
MACRO: loc (*Block* , *N*) $>_{Int}$ loc (*Block* , *N′*) = *N* $>_{Int}$ *N′*
MACRO: loc (*Block* , *N*) $\leq_{Int}$ loc (*Block* , *N′*) = *N* $\leq_{Int}$ *N′*
MACRO: loc (*Block* , *N*) $\geq_{Int}$ loc (*Block* , *N′*) = *N* $\geq_{Int}$ *N′*
MACRO: locations (·) = ·
MACRO: locations (bwrite (*Loc* , —) *L*) = *Loc* locations (*L*)
MACRO: setOfTypes = *Set*(l (arrayType) , l (bitfieldType) , l (functionType) , l (pointerType) , l (structType) , l (unionType) , l (qualifiedType))
MACRO: putInMem (*Loc* , tv (*V* , *T*)) = putInMem-aux (*Loc* , tv (*V* , *T*) , *T* , sizeofType (*T*))
END MODULE

MODULE C-SETTINGS
IMPORTS INCOMING-MODULES
IMPORTS COMMON-SEMANTIC-SYNTAX
Type ::= cfg:sizeut
| cfg:ptrdiffut
Nat ::= rank(*Type*)

K RULES:
RULE: char ⇀ signed-char

RULE: NULL ⇀ tv (loc (0 , 0) , pointerType (void))

MACRO: numBitsPerByte = 8
MACRO: numBytes (signed-char) = 1
MACRO: numBytes (short-int) = 2
MACRO: numBytes (int) = 4
MACRO: numBytes (long-int) = 4
MACRO: numBytes (long-long-int) = 8
MACRO: numBytes (float) = 4
MACRO: numBytes (double) = 8
MACRO: numBytes (long-double) = 16
MACRO: numBytes (enumType (*X*)) = numBytes (int)
MACRO: cfg:sizeut = unsigned-long-int
MACRO: cfg:ptrdiffut = int
MACRO: min (enumType (—)) = min (int)
MACRO: max (enumType (—)) = max (int)
MACRO: sizeofType (*T*) = tv (numBytes (*T*) , cfg:sizeut)
MACRO: sizeofType (pointerType (—)) = tv (numBytes (unsigned-long-int) , cfg:sizeut)
MACRO: rank (char) = 1
MACRO: rank (signed-char) = 1
MACRO: rank (unsigned-char) = 1
MACRO: rank (short-int) = 2
MACRO: rank (unsigned-short-int) = 2
MACRO: rank (int) = 3
MACRO: rank (unsigned-int) = 3
MACRO: rank (long-int) = 4
MACRO: rank (unsigned-long-int) = 4
MACRO: rank (long-long-int) = 5
MACRO: rank (unsigned-long-long-int) = 5
MACRO: rank (enumType (—)) = rank (int)
END MODULE

MODULE SEMANTIC-HELPERS
IMPORTS COMMON-SEMANTIC-SYNTAX
IMPORTS C-SETTINGS
Bool ::= *K* isa *KLabel*

K RULES:
DISCARD RULE: $\langle V \curvearrowright \underline{\text{discard}} \cdots \rangle_k$
.

FIRSTCHAR RULE: firstChar (S) ⇀ substrString (S , 0 , 1)

CHARTOASCII RULE: charToAscii (C) ⇀ asciiString (C)

BUTFIRSTCHAR RULE: butFirstChar (S) ⇀ substrString (S , 1 , lengthString (S))

CALCSTRUCTSIZE RULE: $\langle \underline{\text{calcStructSize (} L \text{)}} \cdots \rangle_k$
necessaryBytes (calcStructSize-aux (L , 0))

CALCUNIONSIZE RULE: $\langle \underline{\text{calcUnionSize (} L \text{)}} \cdots \rangle_k$
necessaryBytes (calcUnionSize-aux (L , 0))

CALCSTRUCTSIZE-HEAT RULE: $\langle \underline{\text{calcStructSize-aux (typedField (} T \text{ , } X \text{) , } L \text{ , } R \text{)}} \cdots \rangle_k$ when T isa bitfieldType
sizeofType (T) ↷ calcStructSize-aux (L , R)

CALCSTRUCTSIZE-HEAT RULE: $\langle \underline{\text{calcStructSize-aux (typedField (} T \text{ , } X \text{) , } L \text{ , } R \text{)}} \cdots \rangle_k$ when $\neg_{Bool}$ T isa bitfieldType
sizeofType (T) ↷ calcStructSize-aux (L , truncRat (R $+_{Rat}$ 7 /Rat 8))

CALCUNIONSIZE-HEAT RULE: $\langle \underline{\text{calcUnionSize-aux (typedField (} T \text{ , } X \text{) , } L \text{ , } R \text{)}} \cdots \rangle_k$
sizeofType (T) ↷ calcUnionSize-aux (L , R)

CALCSTRUCTSIZE-COOL RULE: $\langle \underline{\text{tv (} R' \text{ , —) ↷ calcStructSize-aux (} L \text{ , } R \text{)}} \cdots \rangle_k$
calcStructSize-aux (L , R $+_{Rat}$ R')

CALCUNIONSIZE-COOL RULE: $\langle \underline{\text{tv (} R' \text{ , —) ↷ calcUnionSize-aux (} L \text{ , } R \text{)}} \cdots \rangle_k$
calcUnionSize-aux (L , maxRat (R , R'))

CALCSTRUCTSIZE-DONE RULE: $\langle \underline{\text{calcStructSize-aux (} \cdot List\{K\} \text{ , } R \text{)}} \cdots \rangle_k$
tv (R , cfg:sizeut)

CALCUNIONSIZE-DONE RULE: $\langle \underline{\text{calcUnionSize-aux (} \cdot List\{K\} \text{ , } R \text{)}} \cdots \rangle_k$
tv (R , cfg:sizeut)

MACRO: numBytes (unsigned-char) = numBytes (signed-char)
MACRO: numBytes (unsigned-short-int) = numBytes (short-int)
MACRO: numBytes (unsigned-int) = numBytes (int)
MACRO: numBytes (unsigned-long-int) = numBytes (long-int)
MACRO: numBytes (unsigned-long-long-int) = numBytes (long-long-int)
MACRO: numBytes (registerInt) = numBytes (int)
EQUATION: numBits (T) = numBitsPerByte $*_{Nat}$ numBytes (T) when $getKLabel(T) \neq_{Bool}$ bitfieldType
MACRO: numBits (bitfieldType (— , N)) = N
MACRO: max (registerInt) = max (int)
MACRO: min (registerInt) = min (int)

MACRO: min (signed-char) = $-_{Int}$ 2 ^$_{Nat}$ | numBits (signed-char) $-_{Int}$ 1 |$_{Int}$
MACRO: max (signed-char) = 2 ^$_{Nat}$ | numBits (signed-char) $-_{Int}$ 1 |$_{Int}$ $-_{Int}$ 1
MACRO: min (short-int) = $-_{Int}$ 2 ^$_{Nat}$ | numBits (short-int) $-_{Int}$ 1 |$_{Int}$
MACRO: max (short-int) = 2 ^$_{Nat}$ | numBits (short-int) $-_{Int}$ 1 |$_{Int}$ $-_{Int}$ 1
MACRO: min (int) = $-_{Int}$ 2 ^$_{Nat}$ | numBits (int) $-_{Int}$ 1 |$_{Int}$
MACRO: max (int) = 2 ^$_{Nat}$ | numBits (int) $-_{Int}$ 1 |$_{Int}$ $-_{Int}$ 1
MACRO: min (long-int) = $-_{Int}$ 2 ^$_{Nat}$ | numBits (long-int) $-_{Int}$ 1 |$_{Int}$
MACRO: max (long-int) = 2 ^$_{Nat}$ | numBits (long-int) $-_{Int}$ 1 |$_{Int}$ $-_{Int}$ 1
MACRO: min (long-long-int) = $-_{Int}$ 2 ^$_{Nat}$ | numBits (long-long-int) $-_{Int}$ 1 |$_{Int}$
MACRO: max (long-long-int) = 2 ^$_{Nat}$ | numBits (long-long-int) $-_{Int}$ 1 |$_{Int}$ $-_{Int}$ 1
MACRO: min (unsigned-char) = 0
MACRO: max (unsigned-char) = 2 ^$_{Nat}$ | numBits (unsigned-char) |$_{Int}$ $-_{Int}$ 1
MACRO: min (unsigned-short-int) = 0
MACRO: max (unsigned-short-int) = 2 ^$_{Nat}$ | numBits (unsigned-short-int) |$_{Int}$ $-_{Int}$ 1
MACRO: min (unsigned-int) = 0
MACRO: max (unsigned-int) = 2 ^$_{Nat}$ | numBits (unsigned-int) |$_{Int}$ $-_{Int}$ 1
MACRO: min (unsigned-long-int) = 0
MACRO: max (unsigned-long-int) = 2 ^$_{Nat}$ | numBits (unsigned-long-int) |$_{Int}$ $-_{Int}$ 1
MACRO: min (unsigned-long-long-int) = 0
MACRO: max (unsigned-long-long-int) = 2 ^$_{Nat}$ | numBits (unsigned-long-long-int) |$_{Int}$ $-_{Int}$ 1
MACRO: N to N = $\cdot List\{K\}$
EQUATION: N to N' = N , N +$_{Nat}$ 1 to N' when N <$_{Nat}$ N'
MACRO: S K contains K = true
EQUATION: S K_1 contains K_2 = S contains K_2 when K_1 ≠$_{Bool}$ K_2
MACRO: · contains K = false
MACRO: isArrayType (arrayType (— , —)) = true
EQUATION: isArrayType (T) = false when $getKLabel(T)$ ≠$_{Bool}$ arrayType
MACRO: isFunctionType (functionType (— , —)) = true
EQUATION: isFunctionType (T) = false when $getKLabel(T)$ ≠$_{Bool}$ functionType
EQUATION: isFloatType (T) = true when T =$_{Bool}$ float ∨$_{Bool}$ T =$_{Bool}$ long-double ∨$_{Bool}$ T =$_{Bool}$ double
EQUATION: isIntegerType (T) = true when integerTypes contains T
EQUATION: isIntegerType (T) = false when setOfTypes contains l ($getKLabel(T)$) ∧$_{Bool}$ $getKLabel(T)$ ≠$_{Bool}$ bitfieldType
MACRO: isIntegerType (bitfieldType (— , —)) = true
EQUATION: isUnsignedIntegerType (T) = true when unsignedIntegerTypes contains T
EQUATION: isUnsignedIntegerType (T) = false when setOfTypes contains l ($getKLabel(T)$) ∧$_{Bool}$ $getKLabel(T)$ ≠$_{Bool}$ bitfieldType
EQUATION: isUnsignedIntegerType (bitfieldType (T , —)) = true when isUnsignedIntegerType (T) =$_{Bool}$ true
EQUATION: isUnsignedIntegerType (bitfieldType (T , —)) = false when isUnsignedIntegerType (T) =$_{Bool}$ false
EQUATION: isSignedIntegerType (T) = true when signedIntegerTypes contains T
MACRO: isIntegerType (enumType (—)) = true
EQUATION: isSignedIntegerType (T) = false when setOfTypes contains l ($getKLabel(T)$) ∧$_{Bool}$ $getKLabel(T)$ ≠$_{Bool}$ bitfieldType
EQUATION: isSignedIntegerType (bitfieldType (T , —)) = true when isSignedIntegerType (T) =$_{Bool}$ true
EQUATION: isSignedIntegerType (bitfieldType (T , —)) = false when isSignedIntegerType (T) =$_{Bool}$ false
EQUATION: min (bitfieldType (T , N)) = 0 when unsignedIntegerTypes contains T
EQUATION: max (bitfieldType (T , N)) = 2 ^$_{Nat}$ | N |$_{Int}$ $-_{Int}$ 1 when unsignedIntegerTypes contains T
EQUATION: min (bitfieldType (T , N)) = $-_{Int}$ 2 ^$_{Nat}$ | N $-_{Int}$ 1 |$_{Int}$ when signedIntegerTypes contains T
EQUATION: max (bitfieldType (T , N)) = 2 ^$_{Nat}$ | N $-_{Int}$ 1 |$_{Int}$ $-_{Int}$ 1 when signedIntegerTypes contains T
MACRO: necessaryBytes (tv (R , T)) = tv (truncRat (R +$_{Rat}$ 7 /Rat 8) , T)
MACRO: KL (—) isa KL = true
MACRO: — isa — = false
END MODULE

```
MODULE COMMON-INCLUDE
  IMPORTS INCOMING-MODULES
  IMPORTS SEMANTIC-HELPERS
  IMPORTS C-SETTINGS
  IMPORTS COMMON-SEMANTIC-SYNTAX
END MODULE
```

MODULE COMMON-PARAMETER-BINDING
IMPORTS COMMON-INCLUDE
K ::= bindVariadic(K , $List\{KResult\}$)
| bindVariadic-pre(K , $List\{K\}$)
$List\{K\}$ ::= promoteList($List\{K\}$)

K RULES:
VOID-TO-ID RULE: `Parameter-Declaration` (void) ⇀ ·

RULE: $\langle \frac{\text{bind (tv (} V \text{ , arrayType (} T \text{ , —)) , } L \text{ , } PD \text{)}}{\text{bind (tv (} V \text{ , pointerType (} T \text{)) , } L \text{ , } PD \text{)}} \cdots \rangle_{\text{k}}$

BIND-ONE RULE: $\langle \frac{\text{bind (tv (} V \text{ , } T \text{) , } L \text{ , typedDeclaration (} T' \text{ , } X \text{) , } P \text{)}}{\text{giveLocalType (} T' \text{ , } X \text{)} \curvearrowright \text{addLocals (} Loc \text{)} \curvearrowright \text{storeNew cast (} T' \text{ , tv (} V \text{ , } T \text{)) atLoc } Loc \curvearrowright \text{bind (} L \text{ , } P \text{)}} \cdots \rangle_{\text{k}} \langle \frac{M}{M \text{ [} Loc \text{ / } X \text{]}} \rangle_{\text{env}}$
$\langle \frac{Loc}{\text{inc (} Loc \text{)}} \rangle_{\text{nextLoc}}$

BIND-STRUCT RULE: $\langle \frac{\text{bind (atv (} V \text{ , } T \text{) , } L \text{ , typedDeclaration (} T \text{ , } X \text{) , } P \text{)}}{\text{giveLocalType (} T \text{ , } X \text{)} \curvearrowright \text{storeNew atv (} V \text{ , } T \text{) atLoc } Loc \curvearrowright \text{addLocals (} Loc \text{)} \curvearrowright \text{bind (} L \text{ , } P \text{)}} \cdots \rangle_{\text{k}} \langle \frac{M}{M \text{ [} Loc \text{ / } X \text{]}} \rangle_{\text{env}} \langle \frac{Loc}{\text{inc (} Loc \text{)}} \rangle_{\text{nextLoc}}$

BIND-VARIADIC-PREPARE RULE: $\langle \frac{\text{bind (} L \text{ , typedDeclaration (no-type , ...))}}{\text{bindVariadic-pre (loc (} Block \text{ , } Offset +_{Nat} Len \text{) , promoteList (} L \text{))}} \cdots \rangle_{\text{k}} \langle \cdots \ Block \mapsto \text{memblock (} Len \text{ , —) } \cdots \rangle_{\text{mem}}$
$\langle \text{loc (s}_{Nat} \ Block \text{ , } Offset \text{)} \rangle_{\text{nextLoc}}$

BIND-VARIADIC-START RULE: $\langle \frac{\text{bindVariadic-pre (} N \text{ , } L \text{)}}{\text{bindVariadic (} N \text{ , } L \text{)}} \cdots \rangle_{\text{k}}$

BIND-VARIADIC RULE: $\langle \frac{\text{bindVariadic (} Loc \text{ , } V \text{ , } L \text{)}}{\texttt{sizeof} \text{ (} V \text{)} \curvearrowright \text{bindVariadic (} Loc \text{ , } V \text{ , } L \text{)}} \cdots \rangle_{\text{k}}$

BIND-VARIADIC-DONE RULE: $\langle \frac{\text{bindVariadic (— , } \cdot List\{K\} \text{)}}{\text{sequencePoint}} \cdots \rangle_{\text{k}}$

BIND-VARIADIC-WITHSIZE RULE: $\langle \frac{\text{tv (} Len \text{ , —)} \curvearrowright \text{bindVariadic (loc (} Block \text{ , } Offset \text{) , } V \text{ , } L \text{)}}{\text{append (loc (} Block \text{ , } Offset \text{) , } Len \text{ , } V \text{)} \curvearrowright \text{bindVariadic (loc (} Block \text{ , } Offset +_{Nat} Len \text{) , } L \text{)}} \cdots \rangle_{\text{k}}$

BIND-BAD-PROTOTYPE RULE: $\langle \frac{\text{bind (tv (} V \text{ , } T \text{) , } L \text{ , } \cdot List\{K\} \text{)}}{\text{addLocals (} Loc \text{)} \curvearrowright \text{storeNew tv (} V \text{ , } T \text{) atLoc } Loc \curvearrowright \text{bind (} L \text{ , typedDeclaration (no-type , ...))}} \cdots \rangle_{\text{k}} \langle \frac{Loc}{\text{inc (} Loc \text{)}} \rangle_{\text{nextLoc}}$

EQUATION: promoteList (tv (V , T) , L) = promote (tv (V , T)) , promoteList (L) when isIntegerType (T) $\wedge_{Bool}$ rank (T) $<_{Int}$ rank (int) $\vee_{Bool}$ T $=_{Bool}$ float
EQUATION: promoteList (tv (V , T) , L) = tv (V , T) , promoteList (L) when $\neg_{Bool}$ isIntegerType (T) $\vee_{Bool}$ rank (T) $\geq_{Int}$ rank (int) $\vee_{Bool}$ T $=_{Bool}$ double $\vee_{Bool}$ T $=_{Bool}$ long-double
MACRO: promoteList (atv (LV , T) , L) = atv (LV , T) , promoteList (L)
MACRO: promoteList ($\cdot List\{K\}$) = $\cdot List\{K\}$
END MODULE

MODULE COMMON-GLOBAL-DECLARATION
IMPORTS COMMON-INCLUDE
K ::= initialize(K , K)
| initList(K , $List\{K\}$)
| computeStructType(K , $Type$, $List\{K\}$)

K RULES:

RULE: $\langle \frac{\texttt{Global}\,(\,\text{typedDeclaration}\,(\,T\,,\,X\,)\,)}{\text{giveGlobalType}\,(\,T\,,\,X\,) \curvearrowright \text{alloc}\,(\,Loc\,,\,\text{sizeofType}\,(\,T\,)\,)} \cdots\rangle_{\text{k}}\ \langle \cdots \frac{\cdot}{X \mapsto Loc} \cdots\rangle_{\text{env}}\ \langle \cdots \frac{\cdot}{X \mapsto Loc} \cdots\rangle_{\text{genv}}\ \langle \frac{Loc}{\text{inc}\,(\,Loc\,)} \rangle_{\text{nextLoc}}$ when $getKLabel(T) \neq_{Bool} \text{functionType}$

RULE: $\langle \frac{\texttt{Local}\,(\,\text{typedDeclaration}\,(\,T\,,\,X\,)\,)}{\text{giveLocalType}\,(\,T\,,\,X\,) \curvearrowright \text{alloc}\,(\,Loc\,,\,\text{sizeofType}\,(\,T\,)\,) \curvearrowright \text{addLocals}\,(\,Loc\,)} \cdots\rangle_{\text{k}}\ \langle \cdots \frac{\cdot}{X \mapsto Loc} \cdots\rangle_{\text{env}}\ \langle \frac{Loc}{\text{inc}\,(\,Loc\,)} \rangle_{\text{nextLoc}}$ when $T \neq_{Bool} \text{registerInt}$

RULE: $\langle \frac{\texttt{Local}\,(\,\text{typedDeclaration}\,(\,\text{registerInt}\,,\,X\,)\,)}{\text{giveLocalType}\,(\,\text{registerInt}\,,\,X\,)} \cdots\rangle_{\text{k}}\ \langle \frac{M}{M\,[\,0\,/\,X\,]} \rangle_{\text{registers}}$

FUNCTION-PROTOTYPE RULE: $\langle \frac{\texttt{Global}\,(\,\text{typedDeclaration}\,(\,\text{functionType}\,(\,\text{—}\,,\,\text{—}\,)\,,\,\text{—}\,)\,)}{\cdot} \cdots\rangle_{\text{k}}$

GLOBAL-VARIABLE-DECLARATION-INIT RULE: $\langle \frac{\text{typedDeclaration}\,(\,T\,,\,X\,) = E}{\texttt{Global}\,(\,\text{typedDeclaration}\,(\,T\,,\,X\,)\,) \curvearrowright \text{initialize}\,(\,X\,,\,E\,) \curvearrowright \text{skipval}} \cdots\rangle_{\text{k}}$

RULE: $\langle \frac{\text{initialize}\,(\,K\,,\,\texttt{InitList}\,(\,L\,)\,)}{\text{initList}\,(\,K\,,\,L\,)} \cdots\rangle_{\text{k}}$

RULE: $\langle \frac{\text{initialize}\,(\,K\,,\,\texttt{InitItem}\,(\,E\,)\,)}{*\,\&\,K = E\,;} \cdots\rangle_{\text{k}}$

RULE: $\langle \frac{\text{initList}\,(\,K\,,\,\texttt{Designation}\,(\,\texttt{ArrayDesignator}\,(\,N\,)\,,\,E\,)\,,\,L\,)}{\text{initialize}\,(\,K\,[\,N\,]\,,\,E\,) \curvearrowright \text{initList}\,(\,K\,,\,L\,)} \cdots\rangle_{\text{k}}$

RULE: $\langle \frac{\text{initList}\,(\,K\,,\,\texttt{Designation}\,(\,\texttt{FieldDesignator}\,(\,F\,)\,,\,E\,)\,,\,L\,)}{\text{initialize}\,(\,K\,.\,F\,,\,E\,) \curvearrowright \text{initList}\,(\,K\,,\,L\,)} \cdots\rangle_{\text{k}}$

RULE: $\langle \frac{\text{initList}\,(\,K\,,\,\cdot List\{K\}\,)}{\cdot} \cdots\rangle_{\text{k}}$

FUNCTION-DEFINITION RULE: $\langle \frac{\text{typedDeclaration}\,(\,T\,,\,X\,)\{B\,\}}{\text{storeNew Closure}\,(\,X\,,\,T\,,\,B\,)\ \text{atLoc}\ Loc \curvearrowright \text{calculateGotoMap}\,(\,X\,,\,B\,) \curvearrowright \text{skipval}} \cdots\rangle_{\text{k}}\ \langle \cdots \frac{\cdot}{X \mapsto Loc} \cdots\rangle_{\text{env}}\ \langle \cdots \frac{\cdot}{X \mapsto Loc} \cdots\rangle_{\text{genv}}\ \langle \cdots \frac{\cdot}{X \mapsto T} \cdots\rangle_{\text{types}}\ \langle \cdots \frac{\cdot}{X \mapsto T} \cdots\rangle_{\text{typedefs}}\ \langle \frac{Loc}{\text{inc}\,(\,Loc\,)} \rangle_{\text{nextLoc}}$

RULE: $\langle \frac{\texttt{Global}\,(\,K_1 \curvearrowright K_2\,)}{\texttt{Global}\,(\,K_1\,) \curvearrowright \texttt{Global}\,(\,K_2\,)} \cdots\rangle_{\text{k}}$

RULE: $\langle \frac{\texttt{enum}\,(\,X\,,\,X' = E\,,\,L\,)}{\texttt{Global}\,(\,\text{Declaration}\,(\,\text{int}\,,\,X'\,) = \texttt{InitItem}\,(\,E\,)\,) \curvearrowright \texttt{enum}\,(\,X\,,\,L\,)} \cdots\rangle_{\text{k}}$

RULE: $\langle \frac{\texttt{enum}\,(\,X\,,\,\cdot\,)}{\text{definedType}\,(\,\text{enumType}\,(\,X\,)\,,\,\text{enum}\,(\,X\,)\,)} \cdots\rangle_{\text{k}}$

RULE: $\langle \underline{\texttt{Field}\,(\,T\,,\,X\,)} \cdots \rangle_{\text{k}}$
$\underline{\text{typedField}\,(\,T\,,\,X\,)}$

CONTEXT: computeStructType (— , — , — , □ , —)
RULE: $\langle \underline{\text{structDef}\,(\,X\,)} \cdots \rangle_{\text{k}}$
$\underline{\text{definedType}\,(\,\text{structType}\,(\,X\,)\,,\,\texttt{struct}\,(\,X\,)\,)}$

RULE: $\langle \underline{\text{unionDef}\,(\,X\,)} \cdots \rangle_{\text{k}}$
$\underline{\text{definedType}\,(\,\text{unionType}\,(\,X\,)\,,\,\texttt{union}\,(\,X\,)\,)}$

RULE: $\langle \underline{\texttt{structDef}\,(\,X\,,\,K\,)} \cdots \rangle_{\text{k}}$
$\underline{\text{giveGlobalType}\,(\,\text{structType}\,(\,X\,)\,,\,\texttt{struct}\,(\,X\,)\,) \curvearrowright \text{computeStructType}\,(\,\texttt{struct}\,(\,X\,)\,,\,\text{structType}\,(\,X\,)\,,\,K\,)}$

RULE: $\langle \underline{\texttt{unionDef}\,(\,X\,,\,K\,)} \cdots \rangle_{\text{k}}$
$\underline{\text{giveGlobalType}\,(\,\text{unionType}\,(\,X\,)\,,\,\texttt{union}\,(\,X\,)\,) \curvearrowright \text{computeStructType}\,(\,\texttt{union}\,(\,X\,)\,,\,\text{unionType}\,(\,X\,)\,,\,K\,)}$

RULE: $\langle \underline{\text{computeStructType}\,(\,K\,,\,T\,,\,L\,)} \cdots \rangle_{\text{k}}$ $\langle \cdots \underline{\cdot} \cdots \rangle_{\text{structs}}$
$\underline{\text{definedType}\,(\,T\,,\,K\,)}$ $\underline{K \mapsto L}$

END MODULE

MODULE COMMON-LOCAL-DECLARATION
IMPORTS COMMON-INCLUDE

K RULES:

RULE: $\langle \underbrace{\mathsf{addLocals}\ (\ \mathit{Loc}\ )}_{\cdot} \cdots \rangle_{\mathsf{k}}\ \langle \cdots \underbrace{\cdot}_{\mathit{Loc}} \cdots \rangle_{\mathsf{locals}}$

END MODULE

MODULE MEMORY

IMPORTS COMMON-INCLUDE

K ::= skipme
| readFromMem-aux(*Nat* , *Type* , K , *List{K}*) [strict(3)]
| dontCountWrite(*Nat*)
| putByteInMem(*Nat* , *BaseValue*)
| putBitInMem(*BaseValue* , *BaseValue*)

Bool ::= isLocation(K)
| known(*Nat*)

K RULES:

RULE: $\langle \underline{\text{readFromMem}(N, T)} \cdots \rangle_{\text{k}}$ $\underline{\text{readFromMem-aux}(N, T, \text{sizeofType}(T), \cdot List\{K\}) \curvearrowright \text{skipme}}$

RULE: $\text{readFromMem-aux}(\text{—}, T, \text{tv}(0, \text{—}), L) \rightharpoonup \text{concretize}(\text{atv}(L, T))$

RULE: $\langle \underline{\text{alloc}(\text{loc}(Block, \text{—}), \text{tv}(Len, \text{—}))} \cdots \rangle_{\text{k}}$ $\langle \cdots \underline{\cdot} \cdots \rangle_{\text{mem}}$ $\underline{Block \mapsto \text{memblock}(Len, \cdot)}$

RULE: $\langle \underline{\text{allocWithDefault}(\text{loc}(Block, \text{—}), \text{tv}(Len, \text{—}), N)} \cdots \rangle_{\text{k}}$ $\langle \cdots \underline{\cdot} \cdots \rangle_{\text{mem}}$ $\underline{Block \mapsto \text{memblock}(Len, 0 \text{ to } Len \mapsto \text{piece}(N, \text{numBitsPerByte}))}$

STORE-CLOSURE RULE: $\langle \underline{\text{store Closure}(K_1, K_2, K_3) \text{ atLoc loc}(Block, Offset)} \cdots \rangle_{\text{k}}$ $\langle \cdots Block \mapsto \text{memblock}(Len, \underline{M}) \cdots \rangle_{\text{mem}}$ $\underline{M[\text{Closure}(K_1, K_2, K_3) / Offset]}$

APPEND-EXISTING RULE: $\langle \underline{\text{append}(\text{loc}(Block, Offset), Len, V)} \cdots \rangle_{\text{k}}$ $\langle \cdots Block \mapsto \underline{\text{memblock}(OldLen, M)} \cdots \rangle_{\text{mem}}$ $\underline{\text{store } V \text{ atLoc loc}(Block, Offset)}$ $\underline{\text{memblock}(Len +_{Nat} OldLen, M)}$

STORE-SCALAR RULE: $\langle \underline{\text{store tv}(V, T) \text{ atLoc } Loc} \cdots \rangle_{\text{k}}$ $\underline{\text{putInMem}(Loc, \text{tv}(V, T))}$

STORE-COMPOSITE RULE: $\langle \underline{\text{store atv}(V, T) \text{ atLoc } Loc} \cdots \rangle_{\text{k}}$ $\underline{\text{putBytesInMem}(Loc, V, T, \text{sizeofType}(T))}$

STORENEW RULE: $\langle \underline{\text{storeNew } V \text{ atLoc } Loc} \cdots \rangle_{\text{k}}$ $\underline{\text{alloc}(Loc, \texttt{sizeof}(V)) \curvearrowright \text{store } V \text{ atLoc } Loc}$

STORE-N-NEW RULE: $\langle \underline{\text{store } Len \text{ New tv}(N, \text{—}) \text{ atLoc } Loc} \cdots \rangle_{\text{k}}$ $\underline{\text{allocWithDefault}(Loc, Len, N)}$

READ-BYTE RULE:
$$\left(\begin{array}{l} \langle \underline{\text{readFromMem-aux}(\text{loc}(Block, \underline{Offset}), T, \text{tv}(\underline{\text{s}_{Nat}\ Left}, T'), L \text{ , } \underline{\cdot List\{K\}})} \cdots \rangle_{\text{k}} \\ \langle \cdots Block \mapsto \text{memblock}(\text{—}, \text{—}, \underline{Offset} \mapsto \underline{V}) \cdots \rangle_{\text{mem}} \ \langle Locs \rangle_{\text{locsWrittenTo}} \ \langle Mem \rangle_{\text{buffer}} \end{array} \right)$$
when $\neg_{Bool}$ loc(*Block* , *Offset*) in $Locs$ $\wedge_{Bool}$ $\neg_{Bool}$ loc(*Block* , *Offset*) in locations(*Mem*)

READ-BYTE-BUFFER RULE: $\langle \underline{\text{readFromMem-aux}(Loc, T, \text{tv}(\text{s}_{Nat}\ Left, T'), L)} \cdots \rangle_{\text{k}}$ $\langle Locs \rangle_{\text{locsWrittenTo}}$ $\langle \cdots \text{bwrite}(Loc, V)\ Mem \rangle_{\text{buffer}}$ when $\underline{\text{readFromMem-aux}(\text{s}_{Nat}\ Loc, T, \text{tv}(Left, T'), L \text{ , } V)}$ $\neg_{Bool}$ Loc in $Locs$ $\wedge_{Bool}$ $\neg_{Bool}$ Loc in locations(*Mem*)

RULE: $\langle \underline{\text{readFromMem-aux}(\text{loc}(Block, \underline{Offset}), T, \text{tv}(\underline{\text{s}_{Nat}\ Left}, T'), \underline{L})} \cdots \rangle_{\text{k}}$ $\langle \cdots Block \mapsto \text{memblock}(BlockLen, \underline{M}) \cdots \rangle_{\text{mem}}$ $\underline{L \text{ , piece}(\text{unknown}(Fresh), \text{numBitsPerByte})}$ $\underline{M[\text{piece}(\text{unknown}(Fresh), \text{numBitsPerByte}) / Offset]}$

$\langle Locs \rangle_{\text{locsWrittenTo}}$ $\langle \underset{\mathsf{s}_{Nat}\ Fresh}{Fresh} \rangle_{\text{freshNat}}$ $\langle Mem \rangle_{\text{buffer}}$ when $\neg_{Bool}$ $\$hasMapping(M , Offset)$ $\wedge_{Bool}$ $\neg_{Bool}$ loc ($Block$, $Offset$) in $Locs$ $\wedge_{Bool}$ $\neg_{Bool}$ loc ($Block$, $Offset$) in locations (Mem) $\wedge_{Bool}$ $Offset$ $<_{Nat}$ $BlockLen$

RULE: $\langle \underset{\cdot}{\text{dontCountWrite (} Loc \text{)}} \cdots \rangle_{\text{k}}$ $\langle \cdots \underset{\cdot}{Loc} \cdots \rangle_{\text{locsWrittenTo}}$

ALLOC-STRING RULE: $\langle \underset{\text{putByteInMem (} Loc \text{ , charToAscii (firstChar (} S \text{))) } \curvearrowright \text{ dontCountWrite (} Loc \text{) } \curvearrowright \text{ allocString (s}_{Nat}\ Loc \text{ , butFirstChar (} S \text{))}}{\text{allocString (} Loc \text{ , } S \text{)}} \cdots \rangle_{\text{k}}$ when lengthString (S) $>_{Int}$ 0

ALLOC-EMPTY-STRING RULE: $\langle \underset{\text{putByteInMem (} Loc \text{ , 0) } \curvearrowright \text{ dontCountWrite (} Loc \text{)}}{\text{allocString (} Loc \text{ , “”)}} \cdots \rangle_{\text{k}}$

RULE: $\langle \underset{\text{putBytesInMem (} Loc \text{ , piece (} N \text{ , } BitLen \text{) , piece (} N' \text{ , 1) , } L \text{ , } T \text{ , tv (s}_{Nat}\ Len \text{ , } J \text{))}}{\text{putBytesInMem (} Loc \text{ , piece (} N \mid_{Nat} N' \ll_{Nat} BitLen \text{ , s}_{Nat}\ BitLen \text{) , } L \text{ , } T \text{ , tv (s}_{Nat}\ Len \text{ , } J \text{))}} \cdots \rangle_{\text{k}}$ when $BitLen$ $<_{Nat}$ 8

RULE: $\langle \underset{\text{putBytesInMem (loc (} Block \text{ , s}_{Nat}\ Offset \text{) , } L \text{ , } T \text{ , tv (} Len \text{ , } J \text{))}}{\text{putBytesInMem (loc (} Block \text{ , } Offset \text{) , } V \text{ , } L \text{ , } T \text{ , tv (s}_{Nat}\ Len \text{ , } J \text{))}} \cdots \rangle_{\text{k}}$ $\langle \cdots \underset{\text{bwrite (loc (} Block \text{ , } Offset \text{) , } V \text{)}}{\cdot} \rangle_{\text{buffer}}$ $\langle Locs \underset{\text{loc (} Block \text{ , } Offset \text{)}}{\cdot} \rangle_{\text{locsWrittenTo}}$ when $\neg_{Bool}$ loc ($Block$, $Offset$) in $Locs$

RULE: $\langle \text{false} \rangle_{\text{blocked}}$ $\langle \underset{\cdot}{\text{bwrite (loc (} Block \text{ , } Offset \text{) , } V \text{)}} \cdots \rangle_{\text{buffer}}$ $\langle \cdots Block \mapsto \text{memblock (—} , \underset{M \text{ [} V \text{ / } Offset \text{]}}{M} \text{)} \cdots \rangle_{\text{mem}}$

RULE: $\langle \underset{\cdot}{\text{putBytesInMem (} Loc \text{ , } \cdot List\{K\} \text{ , } T \text{ , tv (0 , —))}} \cdots \rangle_{\text{k}}$

RULE: $\langle \underset{\cdot}{\text{putInMem-aux (— , — , — , tv (0 , —))}} \cdots \rangle_{\text{k}}$

RULE: $\langle \underset{\text{putInMem-aux (} Loc \text{ , tv (} N \text{ , pointerType (functionType (} T \text{ , } L \text{))) , pointerType (functionType (} T \text{ , } L' \text{)) , tv (} Len \text{ , } J \text{))}}{\text{putInMem-aux (} Loc \text{ , tv (} N \text{ , pointerType (functionType (} T \text{ , } L' \text{))) , pointerType (functionType (} T \text{ , } L' \text{)) , tv (} Len \text{ , } J \text{))}} \cdots \rangle_{\text{k}}$ when $L \neq_{Bool} L'$

RULE: $\langle \underset{\text{putByteInMem (loc (} Block \text{ , } Offset \text{) , 256 } +_{Int} N \text{ \%}_{Int} \text{ 256 \%}_{Int} \text{ 256) } \curvearrowright \text{ putInMem-aux (loc (} Block \text{ , s}_{Nat}\ Offset \text{) , tv (} N \gg_{Int} 8 \text{ , } T \text{) , } T \text{ , tv (} Len \text{ , } J \text{))}}{\text{putInMem-aux (loc (} Block \text{ , } Offset \text{) , tv (} N \text{ , } T \text{) , } T \text{ , tv (s}_{Nat}\ Len \text{ , } J \text{))}} \cdots \rangle_{\text{k}}$ when $\neg_{Bool}$ isLocation (N)

RULE: $\langle \underset{\cdot}{\text{putInMem-aux (} Target \text{ , tv (} F \text{ , } T \text{) , } T \text{ , —)}} \cdots \rangle_{\text{k}}$ $\langle \cdots \underset{\text{bwrite (} Target \text{ , } F \text{)}}{\cdot} \rangle_{\text{buffer}}$ $\langle Locs \underset{Target}{\cdot} \rangle_{\text{locsWrittenTo}}$ when $\neg_{Bool}$ $Target$ in $Locs$ $\wedge_{Bool}$ isFloatType (T)

RULE: $\langle \text{putInMem-aux (—} , \underset{\text{tv (0 , } T \text{)}}{\text{tv (loc (0 , 0) , } T \text{)}} \text{ , — , —)} \cdots \rangle_{\text{k}}$

RULE: $\langle \underset{\cdot}{\text{putInMem-aux (} Target \text{ , tv (loc (} Block \text{ , } Offset \text{) , } T \text{) , } T \text{ , —)}} \cdots \rangle_{\text{k}}$ $\langle Locs \underset{Target}{\cdot} \rangle_{\text{locsWrittenTo}}$ $\langle \cdots \underset{\text{bwrite (} Target \text{ , loc (} Block \text{ , } Offset \text{))}}{\cdot} \rangle_{\text{buffer}}$ when $\neg_{Bool}$ $Target$ in $Locs$ $\wedge_{Bool}$ $Block \neq_{Bool} 0$ $\vee_{Bool}$ $Offset \neq_{Bool} 0$

RULE: $\langle \underset{\text{putBitInMem (bitloc (} Block \text{ , } Base \text{ , } Offset \text{) , 2 } +_{Int} N \text{ \%}_{Int} \text{ 2 \%}_{Int} \text{ 2) } \curvearrowright \text{ putInMem-aux (bitloc (} Block \text{ , } Base \text{ , s}_{Nat}\ Offset \text{) , tv (} N \gg_{Int} 1 \text{ , } T \text{) , } T \text{ , tv (} Len \text{ , } J \text{))}}{\text{putInMem-aux (bitloc (} Block \text{ , } Base \text{ , } Offset \text{) , tv (} N \text{ , } T \text{) , } T \text{ , tv (s}_{Nat}\ Len \text{ , } J \text{))}} \cdots \rangle_{\text{k}}$ when $Offset$ $<_{Nat}$ 8

RULE: $\langle$ putInMem-aux (bitloc (*Block* , *Base* , 8) , tv (*N* , *T*) , *T* , tv (s_{Nat} *Len* , *J*)) $\cdots\rangle_{\mathsf{k}}$
putInMem-aux (bitloc (*Block* , s_{Nat} *Base* , 0) , tv (*N* , *T*) , *T* , tv (s_{Nat} *Len* , *J*))

RULE: $\langle$ putByteInMem (loc (*Block* , *Offset*) , *N*) $\cdots\rangle_{\mathsf{k}}$ $\langle$ *Locs* . $\rangle_{\mathsf{locsWrittenTo}}$ $\langle\cdots$. $\rangle_{\mathsf{buffer}}$
loc (*Block* , *Offset*) bwrite (loc (*Block* , *Offset*) , piece (*N* , 8))
when $\neg_{Bool}$ loc (*Block* , *Offset*) in *Locs*

RULE: $\langle$ putBitInMem (bitloc (*Block* , *Base* , —) , —) $\cdots\rangle_{\mathsf{k}}$ $\langle\cdots$ *Block* $\mapsto$ memblock (— , *M*) $\cdots\rangle_{\mathsf{mem}}$ when
M [piece (0 , numBitsPerByte) / *Base*]
$\neg_{Bool}$ $\$hasMapping(M , Base)$

RULE: $\langle$ putBitInMem (bitloc (*Block* , *Base* , *Offset*) , *N*) $\cdots\rangle_{\mathsf{k}}$ $\langle\cdots$ *Block* $\mapsto$ memblock (— , — *Base* $\mapsto$ piece (*Old* , 8)) $\cdots\rangle_{\mathsf{mem}}$
piece (*Old* $\&_{Nat}$ 255 xor_{Nat} 1 $\ll_{Nat}$ *Offset* $|_{Nat}$ *N* $\ll_{Nat}$ *Offset* , 8)
when known (*Old*)

MACRO: $V \curvearrowright$ skipme = *V*
MACRO: inc (loc (*N* , *M*)) = loc (inc (*N*) , *M*)
MACRO: s_{Nat} loc (*N* , *M*) = loc (*N* , s_{Nat} *M*)
MACRO: inc (*N*) = s_{Nat} *N*
MACRO: isLocation (loc (— , —)) = true
MACRO: isLocation (bitloc (— , — , —)) = true
MACRO: isLocation (*K*) = false
MACRO: known (unknown (—)) = false
MACRO: known (—) = true
END MODULE

MODULE COMMON-SEMANTICS-PROMOTIONS
IMPORTS COMMON-INCLUDE
K ::= arithConversion(*KLabel* , K , K) [strict]
| arithConversion-int(*KLabel* , K , K) [strict]
Bool ::= isArithBinOp(*KLabel*)
| isArithBinConversionOp(*KLabel*)
Set ::= arithBinOps [strat(0)memo]
| arithBinConversionOps [strat(0)memo]
Type ::= maxType(*Type* , *Type*)

K RULES:

RULE: $\langle L\ (\ \frac{\text{tv}\ (\ V\ ,\ T\ )}{\text{promote}\ (\ \text{tv}\ (\ V\ ,\ T\ )\ )}\ ,\ \text{—}\ )\ \cdots\rangle_{\text{k}}$ when isIntegerType (T) $\wedge_{Bool}$ isArithBinOp (L) $\wedge_{Bool}$ rank (T) $<_{Int}$ rank (int)

RULE: $\langle L\ (\ \text{—}\ ,\ \frac{\text{tv}\ (\ V\ ,\ T\ )}{\text{promote}\ (\ \text{tv}\ (\ V\ ,\ T\ )\ )}\ )\ \cdots\rangle_{\text{k}}$ when isIntegerType (T) $\wedge_{Bool}$ isArithBinOp (L) $\wedge_{Bool}$ rank (T) $<_{Int}$ rank (int)

RULE: $\langle\ \frac{L\ (\ \text{tv}\ (\ V\ ,\ T\ )\ ,\ \text{tv}\ (\ V'\ ,\ T'\ )\ )}{\text{arithConversion-int}\ (\ L\ ,\ \text{tv}\ (\ V\ ,\ T\ )\ ,\ \text{tv}\ (\ V'\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when isIntegerType (T) $\wedge_{Bool}$ isIntegerType (T') $\wedge_{Bool}$ isArithBinConversionOp (L) $\wedge_{Bool}$ rank (T) $\geq_{Int}$ rank (int) $\wedge_{Bool}$ rank (T') $\geq_{Int}$ rank (int) $\wedge_{Bool}$ $T \neq_{Bool} T'$

RULE: promote (tv (V , T)) $\rightharpoonup$ tv (V , int) when isIntegerType (T) $\wedge_{Bool}$ min (int) $\leq_{Int}$ min (T) $\wedge_{Bool}$ rank (T) $\leq_{Int}$ rank (int) $\wedge_{Bool}$ max (int) $\geq_{Int}$ max (T)

RULE: promote (tv (V , T)) $\rightharpoonup$ cast (unsigned-int , tv (V , T)) when $\neg_{Bool}$ min (int) $\leq_{Int}$ min (T) $\wedge_{Bool}$ max (int) $\geq_{Int}$ max (T) $\wedge_{Bool}$ isIntegerType (T) $\wedge_{Bool}$ rank (T) $\leq_{Int}$ rank (int)

RULE: promote (tv (F , float)) $\rightharpoonup$ tv (F , double)

RULE: $\langle \frac{\text{arithConversion-int}\ (\ L\ ,\ \text{tv}\ (\ V\ ,\ T\ )\ ,\ \text{tv}\ (\ V'\ ,\ T\ )\ )}{L\ (\ \text{tv}\ (\ V\ ,\ T\ )\ ,\ \text{tv}\ (\ V'\ ,\ T\ )\ )}\ \cdots\rangle_{\text{k}}$

RULE: $\langle\ \frac{\text{arithConversion-int}\ (\ L\ ,\ \text{tv}\ (\ V\ ,\ T\ )\ ,\ \text{tv}\ (\ V'\ ,\ T'\ )\ )}{L\ (\ \text{cast}\ (\ \text{maxType}\ (\ T\ ,\ T'\ )\ ,\ \text{tv}\ (\ V\ ,\ T\ )\ )\ ,\ \text{cast}\ (\ \text{maxType}\ (\ T\ ,\ T'\ )\ ,\ \text{tv}\ (\ V'\ ,\ T'\ )\ )\ )}\ \cdots\rangle_{\text{k}}$ when isUnsignedIntegerType (T) $\wedge_{Bool}$ isUnsignedIntegerType (T') $\vee_{Bool}$ isSignedIntegerType (T) $\wedge_{Bool}$ isSignedIntegerType (T') $\wedge_{Bool}$ $T \neq_{Bool} T'$

MACRO: arithBinOps = arithBinConversionOps *Set*(l (_«_) , l (_»_))
MACRO: arithBinConversionOps = *Set*(l (_*_) , l (_/_) , l (_%_) , l (_+_) , l (_-_) , l (_<_) , l (_>_) , l (_<=_) , l (_>=_) , l (_==_) , l (_!=_) , l (_&_) , l (_|_))
MACRO: isArithBinOp (KL) = if arithBinOps contains l (KL) then true else false fi
MACRO: isArithBinConversionOp (KL) = if arithBinConversionOps contains l (KL) then true else false fi
EQUATION: maxType (T , T') = T when rank (T) $\geq_{Int}$ rank (T')
EQUATION: maxType (T , T') = T' when rank (T') $\geq_{Int}$ rank (T)
END MODULE

MODULE COMMON-SEMANTICS-CAST
IMPORTS COMMON-INCLUDE

K RULES:

RULE: $\texttt{Cast}\ (\ T\ ,\ E\ ) \rightharpoonup \text{cast}\ (\ T\ ,\ E\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ B\ ,\ T\ )\ )}\ \cdots\rangle_{\text{k}}$
$\text{tv}\ (\ B\ ,\ T\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T\ ) \wedge_{Bool} \text{isIntegerType}\ (\ T'\ ) \wedge_{Bool} \text{min}\ (\ T\ ) \leq_{Int} I \wedge_{Bool} \text{max}\ (\ T\ ) \geq_{Int} I$
$\text{tv}\ (\ I\ ,\ T\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T'\ ) \wedge_{Bool} I <_{Int} \text{min}\ (\ T\ ) \wedge_{Bool} \text{unsignedIntegerTypes contains}\ T$
$\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I +_{Int} 1 +_{Int} \text{max}\ (\ T\ )\ ,\ T'\ )\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T'\ ) \wedge_{Bool} I >_{Int} \text{max}\ (\ T\ ) \wedge_{Bool} \text{unsignedIntegerTypes contains}\ T$
$\text{tv}\ (\ I\ \%_{Int}\ 1 +_{Int} \text{max}\ (\ T\ )\ ,\ T\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\neg_{Bool} \text{unsignedIntegerTypes contains}\ T \wedge_{Bool} \text{isIntegerType}\ (\ T\ ) \wedge_{Bool} \text{isIntegerType}\ (\ T'\ ) \wedge_{Bool}$
$I <_{Int} \text{min}\ (\ T\ )$
$\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I +_{Int} 2\ \hat{}_{Nat}\ |\ \text{numBits}\ (\ T\ )\ |_{Int}\ ,\ T'\ )\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\neg_{Bool} \text{unsignedIntegerTypes contains}\ T \wedge_{Bool} \text{isIntegerType}\ (\ T\ ) \wedge_{Bool} \text{isIntegerType}\ (\ T'\ ) \wedge_{Bool}$
$I >_{Int} \text{max}\ (\ T\ )$
$\text{cast}\ (\ T\ ,\ \text{tv}\ (\ I -_{Int} 2\ \hat{}_{Nat}\ |\ \text{numBits}\ (\ T\ )\ |_{Int}\ ,\ T'\ )\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ V\ ,\ \text{double}\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T\ )$
$\text{cast}\ (\ T\ ,\ \text{tv}\ (\ \text{truncRat}\ (\ \text{Float2Rat}\ (\ V\ )\ )\ ,\ \text{long-long-int}\ )\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ V\ ,\ \text{float}\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T\ )$
$\text{cast}\ (\ T\ ,\ \text{tv}\ (\ \text{truncRat}\ (\ \text{Float2Rat}\ (\ V\ )\ )\ ,\ \text{long-long-int}\ )\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ V\ ,\ \text{long-double}\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T\ )$
$\text{cast}\ (\ T\ ,\ \text{tv}\ (\ \text{truncRat}\ (\ \text{Float2Rat}\ (\ V\ )\ )\ ,\ \text{long-long-int}\ )\ )$

RULE: $\langle \underline{\text{cast}\ (\ T'\ ,\ \text{tv}\ (\ I\ ,\ T\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T\ ) \wedge_{Bool} T' =_{Bool} \text{float} \vee_{Bool} T' =_{Bool} \text{double} \vee_{Bool} T' =_{Bool} \text{long-double}$
$\text{tv}\ (\ \text{Rat2Float}\ (\ I\ )\ ,\ T'\ )$

RULE: $\langle \underline{\text{cast}\ (\ T'\ ,\ \text{tv}\ (\ F\ ,\ T\ )\ )}\ \cdots\rangle_{\text{k}}$ when $T =_{Bool} \text{float} \vee_{Bool} T =_{Bool} \text{double} \vee_{Bool} T =_{Bool} \text{long-double} \wedge_{Bool} T' =_{Bool} \text{float} \vee_{Bool} T' =_{Bool} \text{double} \vee_{Bool} T' =_{Bool} \text{long-double}$
$\text{tv}\ (\ F\ ,\ T'\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ \text{loc}\ (\ N\ ,\ M\ )\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T\ ) \wedge_{Bool} \text{isIntegerType}\ (\ T'\ )$
$\text{tv}\ (\ \text{loc}\ (\ N\ ,\ M\ )\ ,\ T\ )$

RULE: $\langle \underline{\text{cast}\ (\ \text{pointerType}\ (\ T\ )\ ,\ \text{tv}\ (\ I\ ,\ T'\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T'\ ) \wedge_{Bool} I \neq_{Bool} 0$
$\text{tv}\ (\ I\ ,\ \text{pointerType}\ (\ T\ )\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ \text{loc}\ (\ N\ ,\ M\ )\ ,\ \text{pointerType}\ (\ T'\ )\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\neg_{Bool} N =_{Bool} 0 \wedge_{Bool} M =_{Bool} 0 \wedge_{Bool} \text{isIntegerType}\ (\ T\ )$
$\text{tv}\ (\ \text{loc}\ (\ N\ ,\ M\ )\ ,\ T\ )$

RULE: $\langle \underline{\text{cast}\ (\ T\ ,\ \text{tv}\ (\ N\ ,\ \text{pointerType}\ (\ T'\ )\ )\ )}\ \cdots\rangle_{\text{k}}$ when $\text{isIntegerType}\ (\ T\ ) \wedge_{Bool} N >_{Nat} 0$
$\text{cast}\ (\ T\ ,\ \text{tv}\ (\ N\ ,\ \text{unsigned-long-int}\ )\ )$

RULE: $\langle \underline{\underline{\mathsf{cast}\ (\ \mathsf{pointerType}\ (\ T\ )\ ,\ \mathsf{tv}\ (\ 0\ ,\ T'\ )\ )}}\ \cdots \rangle_{\mathsf{k}}$ when $\mathsf{isIntegerType}\ (\ T'\ )$
$\mathsf{tv}\ (\ \mathsf{loc}\ (\ 0\ ,\ 0\ )\ ,\ \mathsf{pointerType}\ (\ T\ )\ )$

RULE: $\langle \underline{\underline{\mathsf{cast}\ (\ T\ ,\ \mathsf{tv}\ (\ \mathsf{loc}\ (\ 0\ ,\ 0\ )\ ,\ \mathsf{pointerType}\ (\ T'\ )\ )\ )}}\ \cdots \rangle_{\mathsf{k}}$ when $\mathsf{isIntegerType}\ (\ T\ )$
$\mathsf{tv}\ (\ 0\ ,\ T\ )$

RULE: $\langle \underline{\underline{\mathsf{cast}\ (\ \mathsf{pointerType}\ (\ T\ )\ ,\ \mathsf{tv}\ (\ N\ ,\ \mathsf{pointerType}\ (\ \_\ )\ )\ )}}\ \cdots \rangle_{\mathsf{k}}$
$\mathsf{tv}\ (\ N\ ,\ \mathsf{pointerType}\ (\ T\ )\ )$

END MODULE

MODULE COMMON-C-CONVERSIONS
IMPORTS COMMON-SEMANTICS-CAST
IMPORTS COMMON-SEMANTICS-PROMOTIONS
K ::= concretize-aux(*Value* , *Nat*)

K RULES:

RULE: interpret (T , I) ⇀ tv (I , T) when isIntegerType (T) $\wedge_{Bool}$ min (T) $\leq_{Int}$ I $\wedge_{Bool}$ max (T) $\geq_{Int}$ I

RULE: interpret (T , I) ⇀ interpret (T , $I -_{Int}$ 2 ^$_{Nat}$ | numBits (T) |$_{Int}$) when isIntegerType (T) $\wedge_{Bool}$ $I >_{Int}$ max (T) $\wedge_{Bool}$ $I -_{Int}$ 2 ^$_{Nat}$ | numBits (T) |$_{Int}$ $\geq_{Int}$ min (T)

RULE: arithInterpret (T , I) ⇀ tv (I , T) when isIntegerType (T) $\wedge_{Bool}$ min (T) $\leq_{Int}$ I $\wedge_{Bool}$ max (T) $\geq_{Int}$ I

RULE: tv (I , bitfieldType (T , N)) ⇀ tv (I , T)

RULE: leftShiftInterpret (T , I , tv (E_1 , T)) ⇀ tv (I %$_{Int}$ 1 +$_{Int}$ max (T) , T) when unsignedIntegerTypes contains T

RULE: leftShiftInterpret (T , I , tv (E_1 , T)) ⇀ interpret (T , I) when ¬$_{Bool}$ unsignedIntegerTypes contains T $\wedge_{Bool}$ isIntegerType (T) $\wedge_{Bool}$ $I \leq_{Int}$ 2 ^$_{Nat}$ | numBits (T) |$_{Int}$

RULE: rightShiftInterpret (T , I) ⇀ tv (I , T) when integerTypes contains T

RULE: arithInterpret (T , F) ⇀ tv (F , T) when $T =_{Bool}$ float $\vee_{Bool}$ $T =_{Bool}$ double $\vee_{Bool}$ $T =_{Bool}$ long-double

RULE: arithInterpret (T , I) ⇀ tv (I %$_{Int}$ 1 +$_{Int}$ max (T) , T) when $I >_{Int}$ max (T) $\wedge_{Bool}$ unsignedIntegerTypes contains T

RULE: arithInterpret (T , I) ⇀ arithInterpret (T , $I +_{Int}$ 1 +$_{Int}$ max (T)) when $I <_{Int}$ min (T) $\wedge_{Bool}$ $I +_{Int}$ 1 +$_{Int}$ max (T) $\leq_{Int}$ max (T) $\wedge_{Bool}$ unsignedIntegerTypes contains T

EQUATION: concretize (atv (L , T)) = concretize-aux (atv (L , T) , 0) when isIntegerType (T) $\wedge_{Bool}$ $getKLabel(T) \neq_{Bool}$ pointerType
MACRO: concretize (atv (piece (N , *Len*) , L , pointerType (T))) = concretize-aux (atv (piece (N , *Len*) , L , pointerType (T)) , 0)
MACRO: concretize (atv (F , L , float)) = tv (F , float)
MACRO: concretize (atv (F , L , double)) = tv (F , double)
MACRO: concretize (atv (F , L , long-double)) = tv (F , long-double)
EQUATION: concretize (atv (loc (N , M) , L , T)) = tv (loc (N , M) , T) when ¬$_{Bool}$ $N =_{Bool}$ 0 $\wedge_{Bool}$ $M =_{Bool}$ 0
EQUATION: concretize (atv (loc (0 , 0) , L , T)) = tv (0 , T) when isIntegerType (T)
MACRO: concretize (atv (loc (0 , 0) , L , pointerType (T))) = tv (loc (0 , 0) , T)
MACRO: concretize (atv (piece (0 , 8) , L , pointerType (T))) = tv (loc (0 , 0) , pointerType (T))
MACRO: concretize (atv (L , structType (X))) = atv (L , structType (X))
MACRO: concretize (atv (L , unionType (X))) = atv (L , unionType (X))
MACRO: concretize-aux (atv (L , piece (N' , *Len*) , T) , N) = concretize-aux (atv (L , T) , $N' |_{Nat} N \ll_{Nat} Len$)
EQUATION: concretize-aux (atv (*·List{K}* , T) , N) = interpret (T , N) when $getKLabel(T) \neq_{Bool}$ pointerType
MACRO: concretize-aux (atv (*·List{K}* , pointerType (T)) , 0) = tv (loc (0 , 0) , pointerType (T))
EQUATION: concretize-aux (atv (*·List{K}* , pointerType (T)) , N) = tv (N , pointerType (T)) when $N >_{Nat}$ 0
EQUATION: concretize-aux (atv (loc (N , M) , T) , unknown (—) |$_{Nat}$ unknown (—) |$_{Nat}$ unknown (—) $\ll_{Nat}$ 8 $\ll_{Nat}$ 8) = tv (loc (N , M) , T) when ¬$_{Bool}$ $N =_{Bool}$ 0 $\wedge_{Bool}$ $M =_{Bool}$ 0
EQUATION: concretize-aux (atv (loc (0 , 0) , T) , unknown (—) |$_{Nat}$ unknown (—) |$_{Nat}$ unknown (—) $\ll_{Nat}$ 8 $\ll_{Nat}$ 8) = tv (0 , T) when $getKLabel(T) \neq_{Bool}$ pointerType
EQUATION: concretize-aux (atv (loc (N , M) , T) , 0) = tv (loc (N , M) , T) when ¬$_{Bool}$ $N =_{Bool}$ 0 $\wedge_{Bool}$ $M =_{Bool}$ 0
EQUATION: concretize-aux (atv (loc (0 , 0) , T) , 0) = tv (0 , T) when $getKLabel(T) \neq_{Bool}$ pointerType
MACRO: integerTypes = signedIntegerTypes unsignedIntegerTypes
MACRO: unsignedIntegerTypes = *Set*(unsigned-char , unsigned-short-int , unsigned-int , unsigned-long-int , unsigned-long-long-int)
MACRO: signedIntegerTypes = *Set*(char , signed-char , short-int , int , long-int , long-long-int)
END MODULE

MODULE COMMON-SEMANTICS-TYPE-DECLARATIONS
IMPORTS COMMON-INCLUDE
Set ::= declarators [strat(0)memo]

K RULES:

TYPE-VARIADIC RULE: $\langle \underset{\text{typedDeclaration (no-type , ...)}}{\text{...}} \cdots \rangle_{\text{k}}$

SEMANTIC-DECLARE RULE: $\langle \underset{\text{typedDeclaration (} T \text{ , } X \text{)}}{\text{Declaration (} T \text{ , } X \text{)}} \cdots \rangle_{\text{k}}$

TYPEDEF-DECLARATION RULE: $\langle \underset{\text{definedType (} T \text{ , } X \text{)}}{\texttt{Typedef (} T \text{ , } X \text{)}} \cdots \rangle_{\text{k}}$

GIVE-GLOBAL-TYPE RULE: $\langle \underset{\cdot}{\text{giveGlobalType (} T \text{ , } X \text{)}} \cdots \rangle_{\text{k}} \langle \underset{M \text{ [} T \text{ / } X \text{]}}{M} \rangle_{\text{types}} \langle \underset{M' \text{ [} T \text{ / } X \text{]}}{M'} \rangle_{\text{typedefs}}$

RULE: $\langle \underset{\cdot}{\text{giveLocalType (} T \text{ , } X \text{)}} \cdots \rangle_{\text{k}} \langle \underset{M \text{ [} T \text{ / } X \text{]}}{M} \rangle_{\text{types}}$

GLOBAL-DECLARATION RULE: $\langle \underset{\text{giveGlobalType (} T \text{ , } X \text{)}}{\texttt{Global (definedType (} T \text{ , } X \text{))}} \cdots \rangle_{\text{k}}$

RULE: $\langle \underset{X}{X + I} \cdots \rangle_{\text{type}}$

RULE: $\langle \underset{X}{X - I} \cdots \rangle_{\text{type}}$

RULE: $T \text{ [} N \text{]} \rightharpoonup \text{arrayType (} T \text{ , } N \text{)}$

RULE: $\texttt{Pointer (} T \text{)} \rightharpoonup \text{pointerType (} T \text{)}$

RULE: $\text{Pointer (} T \text{ , } \_ \text{)} \rightharpoonup \text{pointerType (} T \text{)}$

RULE: $\text{Direct-Function-Declarator (} PTL \text{)} \rightharpoonup \texttt{Direct-Function-Declarator (anonymousId , } PTL \text{)}$

RULE: $\langle \underset{\cdot}{\texttt{Global (skipval)}} \cdots \rangle_{\text{k}}$

RULE: $\langle \underset{L \text{ (pointerType (} T \text{) , } D \text{)}}{L \text{ (} T \text{ , } \texttt{Pointer (} D \text{))}} \cdots \rangle_{\text{k}}$ when declarators contains I (L)

RULE: $\langle \underset{L \text{ (functionType (} T \text{ , } PTL \text{) , } D \text{)}}{L \text{ (} T \text{ , } \texttt{Direct-Function-Declarator (} D \text{ , } PTL \text{))}} \cdots \rangle_{\text{k}}$ when declarators contains I (L)

RULE: $\langle L \text{ (} \underset{T}{X} \text{ , } D \text{)} \cdots \rangle_{\text{k}} \langle \cdots \texttt{ typedefName (} X \text{)} \mapsto T \cdots \rangle_{\text{typedefs}}$ when declarators contains I (L)

RULE: $\langle \underset{L \text{ (arrayType (} T \text{ , } N \text{) , } K \text{)}}{L \text{ (} T \text{ , } K \text{ [tv (} N \text{ , } \_ \text{)])}} \cdots \rangle_{\text{k}}$ when declarators contains I (L)

RULE: L (T , BitField (D , N)) ⇀ L (bitfieldType (T , N) , D) when declarators contains l (L)

RULE: L (T , `BitField` (N)) ⇀ L (bitfieldType (T , N) , unnamedBitField) when declarators contains l (L)

MACRO: declarators = *Set*(l (`Declaration`) , l (`Field`) , l (`Parameter-Declaration`) , l (`Typedef`) , l (`Pointer`))

END MODULE

MODULE COMMON-SEMANTICS-TYPE-STRICTNESS
IMPORTS COMMON-INCLUDE

K RULES:

CONTEXT: — (— , `Direct-Function-Declarator` (D , □))

RULE: $\langle \underline{\texttt{Parameter-Type-List}\ (\ V\ )} \cdots \rangle_{\text{k}}$
$\underline{\text{typedParameterList}\ (\ V\ )}$

RULE: $\langle \underline{\text{Parameter-Declaration}\ (\ T\ ,\ X\ )} \cdots \rangle_{\text{k}}$
$\underline{\text{typedDeclaration}\ (\ T\ ,\ X\ )}$

RULE: $\langle \underline{\texttt{Parameter-Declaration}\ (\ T\ )} \cdots \rangle_{\text{k}}$
$\underline{\text{typedDeclaration}\ (\ T\ ,\ \text{anonymousId}\ )}$

CONTEXT: `Field` (— , — [□])
CONTEXT: Declaration (— , — [□])
CONTEXT: Parameter-Declaration (— , — [□])
CONTEXT: `Typedef` (— , — [□])

TYPE-ARROW-HEAT RULE: $\langle \underline{Kp\ \texttt{->}\ X} \cdots \rangle_{\text{type}}$
$\underline{Kp \curvearrowright \text{HOLE}\ \texttt{->}\ X}$

TYPE-ARROW-COOL RULE: $\langle \underline{T \curvearrowright \text{HOLE}\ \texttt{->}\ X} \cdots \rangle_{\text{type}}$
$\underline{T\ \texttt{->}\ X}$

TYPE-DOT-HEAT RULE: $\langle \underline{Kp\ \texttt{.}\ X} \cdots \rangle_{\text{type}}$
$\underline{Kp \curvearrowright \text{HOLE}\ \texttt{.}\ X}$

TYPE-DOT-COOL RULE: $\langle \underline{T \curvearrowright \text{HOLE}\ \texttt{.}\ X} \cdots \rangle_{\text{type}}$
$\underline{T\ \texttt{.}\ X}$

RULE: $\langle \underline{Kp\ [\ E\ ]} \cdots \rangle_{\text{type}}$
$\underline{Kp \curvearrowright \texttt{*}\ \text{HOLE}}$

RULE: $\langle \underline{T \curvearrowright \texttt{*}\ \text{HOLE}} \cdots \rangle_{\text{type}}$
$\underline{\texttt{*}\ T}$

TYPE-HEAT-DEREF RULE: $\langle \underline{\texttt{*}\ Kp} \cdots \rangle_{\text{type}}$
$\underline{Kp \curvearrowright \texttt{*}\ \text{HOLE}}$

END MODULE

MODULE COMMON-C-TYPING
IMPORTS COMMON-SEMANTICS-TYPE-STRICTNESS
IMPORTS COMMON-SEMANTICS-TYPE-DECLARATIONS
K ::= normalizeType(K , K)
| typedef(K , K)
Bool ::= isAType K

K RULES:

RULE: qualifiedType (T , —) ⇀ T

RULE: unsigned-short ⇀ unsigned-short-int

RULE: unsigned-long ⇀ unsigned-long-int

RULE: unsigned-long-long ⇀ unsigned-long-long-int

RULE: short ⇀ short-int

RULE: long ⇀ long-int

RULE: long-long ⇀ long-long-int

RULE: $\langle \underset{\text{evalToType} \curvearrowright \text{typeof (HOLE)}}{\text{typeof (} E \text{)}} \cdots \rangle_{\text{k}}$ $\underset{\langle E \rangle_{\text{type}}}{\cdot}$

RULE: $\langle \underset{T}{\text{evalToType} \curvearrowright \text{typeof (HOLE)}} \cdots \rangle_{\text{k}}$ $\underset{\cdot}{\langle T \rangle_{\text{type}}}$ when isAType T

RULE: $\langle \underset{\text{double}}{F} \rangle_{\text{type}}$

RULE: $\langle \underset{T}{\text{atv (— , } T \text{)}} \rangle_{\text{type}}$

RULE: $\langle \underset{\text{arrayType (char , lengthString (} S \text{))}}{S} \rangle_{\text{type}}$

RULE: $\langle \underset{\vec{T}}{E} \cdots \rangle_{\text{type}}$ $\langle \cdots \; E \mapsto T \; \cdots \rangle_{\text{types}}$

RULE: $\langle \underset{\vec{T}}{X} \cdots \rangle_{\text{type}}$ $\langle \cdots \; \texttt{typedefName} \text{ (} X \text{)} \mapsto T \; \cdots \rangle_{\text{types}}$

RULE: $\langle \underset{T}{\text{tv (— , } T \text{)}} \cdots \rangle_{\text{type}}$

RULE: $\langle \underset{T}{\text{cast (} T \text{ , —)}} \cdots \rangle_{\text{type}}$

TYPE-STRUCT-ARROW RULE: $\langle \underset{T}{\text{pointerType (structType (} S \text{)) -> } X} \cdots \rangle_{\text{type}}$ $\langle \cdots \; \texttt{struct} \text{ (} S \text{)} \mapsto \text{— , typedField (} T \text{ , } X \text{) , —} \; \cdots \rangle_{\text{structs}}$

TYPE-UNION-ARROW RULE: $\langle \underline{\text{pointerType (unionType (} S \text{)) -> } X} \ \cdots \rangle_{\text{type}} \ \langle \cdots \ \texttt{union} \ (\ S\ ) \mapsto \_ \ , \ \text{typedField} \ (\ T\ ,\ X\ )\ ,\ \_ \ \cdots \rangle_{\text{structs}}$
T

TYPE-STRUCT-DOT RULE: $\langle \underline{\text{structType (} S \text{) . } X} \ \cdots \rangle_{\text{type}} \ \langle \cdots \ \texttt{struct} \ (\ S\ ) \mapsto \_ \ , \ \text{typedField} \ (\ T\ ,\ X\ )\ ,\ \_ \ \cdots \rangle_{\text{structs}}$
T

TYPE-UNION-DOT RULE: $\langle \underline{\text{unionType (} S \text{) . } X} \ \cdots \rangle_{\text{type}} \ \langle \cdots \ \texttt{union} \ (\ S\ ) \mapsto \_ \ , \ \text{typedField} \ (\ T\ ,\ X\ )\ ,\ \_ \ \cdots \rangle_{\text{structs}}$
T

RULE: $\langle \underline{\text{Closure (} \_ \text{ , } T \text{ , } \_ \text{)}} \ \cdots \rangle_{\text{type}}$
T

RULE: $\langle \underline{\texttt{Pointer} \ (\ Kp\ )} \ \cdots \rangle_{\text{type}}$
$Kp \curvearrowright \texttt{Pointer} \ (\ \text{HOLE}\ )$

RULE: $\langle \underline{T \curvearrowright \texttt{Pointer} \ (\ \text{HOLE}\ )} \ \cdots \rangle_{\text{type}}$
$\texttt{Pointer} \ (\ T\ )$

TYPE-DEREF-VALUE RULE: $\langle \underline{\text{* tv (} \_ \text{ , pointerType (} T \text{))}} \ \cdots \rangle_{\text{type}}$
T

TYPE-DEREF-TYPE RULE: $\langle \underline{\text{* pointerType (} T \text{)}} \ \cdots \rangle_{\text{type}}$
T

TYPE-DEREF-ARRAY RULE: $\langle \underline{\text{* arrayType (} T \text{ , } \_ \text{)}} \ \cdots \rangle_{\text{type}}$
T

RULE: $\texttt{extern} \rightharpoonup \cdot$

RULE: $\texttt{static} \rightharpoonup \cdot$

EQUATION: isAType T = true when setOfTypes contains l ($getKLabel(T)$)
MACRO: isAType T = true
MACRO: int $\curvearrowright$ `register` = registerInt
MACRO: `register` $\curvearrowright$ int = registerInt
EQUATION: `register` $\curvearrowright$ K = K when $K \neq_{Bool}$ int
EQUATION: $K \curvearrowright$ `register` = K when $K \neq_{Bool}$ int
MACRO: pointerType (registerInt) = pointerType (int)
END MODULE

```
MODULE COMMON-SEMANTICS-EXPRESSIONS-INCLUDE
  IMPORTS COMMON-INCLUDE
  K ::= assign( K , K )
END MODULE
```

MODULE COMMON-SEMANTICS-SIZEOF
IMPORTS COMMON-INCLUDE

K RULES:

RULE: $\langle \underline{\texttt{sizeof}\ (\ E\ )} \cdots \rangle_{\text{k}}$
$\underline{\text{sizeofType}\ (\ \text{typeof}\ (\ E\ )\ )}$

RULE: $\langle \underline{\text{sizeofType}\ (\ \text{arrayType}\ (\ T\ ,\ N\ )\ )} \cdots \rangle_{\text{k}}$
$\underline{\text{sizeofType}\ (\ T\ )\ *\ \text{tv}\ (\ N\ ,\ \text{cfg:sizeut}\ )}$

RULE: $\text{sizeofType}\ (\ \text{bitfieldType}\ (\ T\ ,\ N\ )\ ) \rightharpoonup \text{tv}\ (\ N\ /\text{Rat}\ 8\ ,\ \text{cfg:sizeut}\ )$

RULE: $\text{sizeofType}\ (\ \text{functionType}\ (\ \text{—}\ ,\ \text{—}\ )\ ) \rightharpoonup \text{tv}\ (\ 1\ ,\ \text{cfg:sizeut}\ )$

SIZEOF-STRUCT RULE: $\langle \underline{\text{sizeofType}\ (\ \text{structType}\ (\ X\ )\ )} \cdots \rangle_{\text{k}}\ \langle \cdots\ \texttt{struct}\ (\ X\ ) \mapsto L\ \cdots \rangle_{\text{structs}}$
$\underline{\text{calcStructSize}\ (\ L\ )}$

SIZEOF-UNION RULE: $\langle \underline{\text{sizeofType}\ (\ \text{unionType}\ (\ X\ )\ )} \cdots \rangle_{\text{k}}\ \langle \cdots\ \texttt{union}\ (\ X\ ) \mapsto L\ \cdots \rangle_{\text{structs}}$
$\underline{\text{calcUnionSize}\ (\ L\ )}$

MACRO: $\text{sizeofType}\ (\ \text{qualifiedType}\ (\ T\ ,\ \text{—}\ )\ ) = \text{sizeofType}\ (\ T\ )$

END MODULE

MODULE COMMON-SEMANTICS-IDENTIFIERS
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE

K RULES:

LOOKUP RULE: $\langle \underline{\quad X \quad} \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ X \mapsto Loc\ \cdots \rangle_{\mathsf{env}}\ \langle \cdots\ X \mapsto T\ \cdots \rangle_{\mathsf{types}}$ when $\neg_{Bool}$ isArrayType (T) $\wedge_{Bool}$ $\neg_{Bool}$ isFunctionType (T)
readFromMem (Loc , T)

LOOKUP-REGISTER RULE: $\langle \underline{\quad X \quad} \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ X \mapsto V\ \cdots \rangle_{\mathsf{registers}}\ \langle \cdots\ X \mapsto \mathsf{registerInt}\ \cdots \rangle_{\mathsf{types}}$
tv (V , int)

LOOKUP-ARRAY RULE: $\langle \underline{\quad X \quad} \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ X \mapsto Loc\ \cdots \rangle_{\mathsf{env}}\ \langle \cdots\ X \mapsto \mathsf{arrayType}\ (\ T\ ,\ —\ )\ \cdots \rangle_{\mathsf{types}}$
tv (Loc , pointerType (T))

LOOKUP-CLOSURE RULE: $\langle \underline{\quad X \quad} \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ X \mapsto Loc\ \cdots \rangle_{\mathsf{env}}\ \langle \cdots\ X \mapsto \mathsf{functionType}\ (\ T\ ,\ L\ )\ \cdots \rangle_{\mathsf{types}}$
tv (Loc , pointerType (functionType (T , L)))

END MODULE

MODULE COMMON-SEMANTICS-FUNCTION-CALLS
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE

K RULES:

CONTEXT: $\mathsf{application}\ (\,—\,,\,—\,,\,\square\,,\,—\,)$

RULE: $\mathsf{Apply}\ (\,E\,) \rightharpoonup \mathtt{Apply}\ (\,E\,,\,\cdot\,)$

RULE: $\left\langle \dfrac{\mathtt{Apply}\ (\,E\,,\,L\,)}{\mathsf{application}\ (\,E\,,\,L\,)}\ \cdots \right\rangle_{\mathsf{k}}$

FUNCTION-APPLICATION-PRE RULE: $\left\langle \dfrac{\mathsf{application}\ (\,\mathsf{tv}\ (\,\mathsf{loc}\ (\,Block\,,\,Offset\,)\,,\,\mathsf{pointerType}\ (\,\mathsf{functionType}\ (\,—\,,\,—\,)\,)\,)\,,\,L\,)}{\mathsf{application}\ (\,V\,,\,L\,)}\ \cdots \right\rangle_{\mathsf{k}}\ \left\langle \cdots\ Block \mapsto \mathsf{memblock}\ (\,—\,,\,—\ \ Offset \mapsto V\,)\ \cdots \right\rangle_{\mathsf{mem}}$

FUNCTION-APPLICATION RULE:

$$\left(\begin{array}{c} \left\langle \dfrac{\mathsf{application}\ (\,\mathsf{Closure}\ (\,X\,,\,\mathsf{functionType}\ (\,R\,,\,\mathsf{typedParameterList}\ (\,P\,)\,)\,,\,B\,)\,,\,L\,) \curvearrowright K}{\mathsf{sequencePoint} \curvearrowright \mathsf{bind}\ (\,L\,,\,P\,) \curvearrowright B} \right\rangle_{\mathsf{k}}\ \left\langle \dfrac{Fun}{X} \right\rangle_{\mathsf{currentFunction}}\ \left\langle \dfrac{E}{G} \right\rangle_{\mathsf{env}}\ \langle G \rangle_{\mathsf{genv}}\ \left\langle \dfrac{L}{\cdot} \right\rangle_{\mathsf{locals}}\ \left\langle \dfrac{T}{GT} \right\rangle_{\mathsf{types}}\ \langle GT \rangle_{\mathsf{typedefs}}\ \left\langle \dfrac{LS}{\cdot} \right\rangle_{\mathsf{loopStack}} \\ \left\langle \dfrac{\cdot}{\mathsf{List}\ (\,\langle \langle Fun \rangle_{\mathsf{currentFunction}}\ \langle K \rangle_{\mathsf{continuation}}\ \langle L \rangle_{\mathsf{locals}}\ \langle E \rangle_{\mathsf{env}}\ \langle T \rangle_{\mathsf{types}}\ \langle LS \rangle_{\mathsf{loopStack}} \rangle_{\mathsf{stackFrame}}\,)}\ \cdots \right\rangle_{\mathsf{callStack}} \end{array}\right)$$

END MODULE

MODULE COMMON-SEMANTICS-ARRAY-SUBSCRIPTING
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE

K RULES:

RULE: $\langle \underset{\overset{\ast}{\ }\ E_1 + E_2}{E_1\ [\ E_2\ ]}\ \cdots \rangle_{\text{k}}$

END MODULE

MODULE COMMON-SEMANTICS-LITERALS
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE
Bool ::= inRange(*Int* , *Type*)

K RULES:

CONST-STRING-NOTFOUND RULE: $\langle \frac{\cdot}{\text{alloc}(\mathit{Loc}, 1 +_{Nat} \text{lengthString}(\mathit{S})) \curvearrowright \text{allocString}(\mathit{Loc}, \mathit{S})} \curvearrowright S \cdots \rangle_{\text{k}} \langle \frac{\mathit{Loc}}{\text{inc}(\mathit{Loc})} \rangle_{\text{nextLoc}} \langle M \frac{\cdot}{S \mapsto \text{tv}(\mathit{Loc}, \text{arrayType}(\text{char}, 1 +_{Nat} \text{lengthString}(\mathit{S})))} \rangle_{\text{statics}}$
when $\neg_{Bool}$ $hasMapping(M , S)$

CONST-STRING-FOUND RULE: $\langle \frac{S}{\text{tv}(N, \text{pointerType}(T))} \cdots \rangle_{\text{k}} \langle \cdots S \mapsto \text{tv}(N, \text{arrayType}(T, —)) \cdots \rangle_{\text{statics}}$

RULE: $\langle \frac{I}{\text{if inRange}(I, \text{int}) \text{ then tv}(I, \text{int}) \text{ else if inRange}(I, \text{long-int}) \text{ then tv}(I, \text{long-int}) \text{ else if inRange}(I, \text{long-long-int}) \text{ then tv}(I, \text{long-long-int}) \text{ else tv}(I, \text{no-type}) \text{ fi fi fi}} \cdots \rangle_{\text{k}}$

RULE: $\texttt{U}(I) \rightharpoonup$ if inRange (I , unsigned-int) then tv (I , unsigned-int) else if inRange (I , unsigned-long-int) then tv (I , unsigned-long-int) else if inRange (I , unsigned-long-long-int) then tv (I , unsigned-long-long-int) else tv (I , no-type) fi fi fi

RULE: $\texttt{L}(I) \rightharpoonup$ if inRange (I , long-int) then tv (I , long-int) else if inRange (I , long-long-int) then tv (I , long-long-int) else tv (I , no-type) fi fi

RULE: $\texttt{UL}(I) \rightharpoonup$ if inRange (I , unsigned-long-int) then tv (I , unsigned-long-int) else if inRange (I , unsigned-long-long-int) then tv (I , unsigned-long-long-int) else tv (I , no-type) fi fi

RULE: $\texttt{LL}(I) \rightharpoonup$ if inRange (I , long-long-int) then tv (I , long-long-int) else tv (I , no-type) fi

RULE: $\texttt{ULL}(I) \rightharpoonup$ if inRange (I , unsigned-long-long-int) then tv (I , unsigned-long-long-int) else tv (I , no-type) fi

RULE: $\langle \frac{F}{\text{tv}(F, \text{double})} \cdots \rangle_{\text{k}}$

RULE: $\texttt{L}(F) \rightharpoonup \text{tv}(F, \text{long-double})$

RULE: $\texttt{F}(F) \rightharpoonup \text{tv}(F, \text{float})$

MACRO: $\text{inRange}(I, T) = I \leq_{Int} \text{max}(T) \wedge_{Bool} I \geq_{Int} \text{min}(T)$

END MODULE

MODULE COMMON-SEMANTICS-ASSIGNMENT
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE

K RULES:

CONTEXT: assign (* □ , —)

CONTEXT: assign (— , □)

FOR-ASSIGNMENT RULE: $\langle \frac{E_1 = E_2}{\text{assign (* \& } E_1 \text{ , } E_2 \text{)}} \cdots \rangle_{\text{k}}$

ASSIGN-TIMES RULE: $E_1 \text{ *= } E_2 \rightharpoonup E_1 = E_1 \text{ * } E_2$

ASSIGN-DIVIDE RULE: $E_1 \text{ /= } E_2 \rightharpoonup E_1 = E_1 \text{ / } E_2$

ASSIGN-MOD RULE: $E_1 \text{ \%= } E_2 \rightharpoonup E_1 = E_1 \text{ \% } E_2$

ASSIGN-PLUS RULE: $E_1 \text{ += } E_2 \rightharpoonup E_1 = E_1 \text{ + } E_2$

ASSIGN-SUBTRACT RULE: $E_1 \text{ -= } E_2 \rightharpoonup E_1 = E_1 \text{ - } E_2$

ASSIGN-LSHIFT RULE: $E_1 \text{ «= } E_2 \rightharpoonup E_1 = E_1 \text{ « } E_2$

ASSIGN-RSHIFT RULE: $E_1 \text{ »= } E_2 \rightharpoonup E_1 = E_1 \text{ » } E_2$

ASSIGN-BIT-AND RULE: $E_1 \text{ \&= } E_2 \rightharpoonup E_1 = E_1 \text{ \& } E_2$

ASSIGN-BIT-XOR RULE: $E_1 \text{ ^= } E_2 \rightharpoonup E_1 = E_1 \text{ ^ } E_2$

ASSIGN-BIT-OR RULE: $E_1 \text{ |= } E_2 \rightharpoonup E_1 = E_1 \text{ | } E_2$

RULE: $\langle \frac{\text{assign (* tv (} Loc \text{ , pointerType (} T \text{)) , tv (} V \text{ , } T' \text{))}}{\text{assign (* tv (} Loc \text{ , pointerType (} T \text{)) , cast (} T \text{ , tv (} V \text{ , } T' \text{)))}} \cdots \rangle_{\text{k}}$ when isIntegerType (T) $\wedge_{Bool}$ isIntegerType (T') $\wedge_{Bool}$ $T \neq_{Bool} T'$

RULE: $\langle \frac{\text{assign (* registerLocation (} X \text{) , tv (} V \text{ , int))}}{\text{tv (} V \text{ , int)}} \cdots \rangle_{\text{k}} \ \langle \cdots \ X \mapsto \frac{\_\_}{V} \ \cdots \rangle_{\text{registers}}$

ASSIGN RULE: $\langle \frac{\text{assign (* tv (} Loc \text{ , pointerType (} T \text{)) , tv (} V \text{ , } T \text{))}}{\text{putInMem (} Loc \text{ , tv (} V \text{ , } T \text{))} \curvearrowright \text{tv (} V \text{ , } T \text{)}} \cdots \rangle_{\text{k}}$

ASSIGN-BITFIELD RULE: $\langle \frac{\text{assign (* tv (bitloc (} Block \text{ , } N \text{ , } O \text{) , pointerType (bitfieldType (} T \text{ , } Len \text{))) , tv (} V \text{ , } T \text{))}}{\text{putInMem-aux (bitloc (} Block \text{ , } N \text{ , } O \text{) , tv (} V \text{ , } T \text{) , } T \text{ , tv (} Len \text{ , cfg:sizeut))} \curvearrowright \text{tv (} V \text{ , } T \text{)}} \cdots \rangle_{\text{k}}$

ASSIGN-FP RULE: $\langle \frac{\text{assign (* tv (} Loc \text{ , pointerType (pointerType (functionType (} T \text{ , } L \text{)))) , tv (} V \text{ , pointerType (functionType (} T \text{ , } L' \text{))))}}{\text{putInMem (} Loc \text{ , tv (} V \text{ , pointerType (functionType (} T \text{ , } L \text{))))} \curvearrowright \text{tv (} V \text{ , pointerType (functionType (} T \text{ , } L \text{)))}} \cdots \rangle_{\text{k}}$

ASSIGN-STRUCT RULE: $\langle \frac{\text{assign (* tv (} Loc \text{ , pointerType (} T \text{)) , atv (} L \text{ , } T \text{))}}{\text{putBytesInMem (} Loc \text{ , } L \text{ , } T \text{ , sizeofType (} T \text{)))} \curvearrowright \text{atv (} L \text{ , } T \text{)}} \cdots \rangle_{\text{k}}$

END MODULE

MODULE COMMON-SEMANTICS-BITWISE
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE

K RULES:

RULE: tv (I , T) « tv (N , T') ⇀ leftShiftInterpret (T , I $\ll_{Int}$ N , tv (I , T)) when isIntegerType (T) $\wedge_{Bool}$ isIntegerType (T') $\wedge_{Bool}$ N $<_{Int}$ numBits (T)

RULE: tv (I , T) » tv (N , T') ⇀ rightShiftInterpret (T , I $\gg_{Int}$ N) when isIntegerType (T) $\wedge_{Bool}$ isIntegerType (T') $\wedge_{Bool}$ N $<_{Int}$ numBits (T)

RULE: tv (I_1 , T) | tv (I_2 , T) ⇀ arithInterpret (T , I_1 $|_{Int}$ I_2) when isIntegerType (T)

RULE: tv (I_1 , T) & tv (I_2 , T) ⇀ arithInterpret (T , I_1 $\&_{Int}$ I_2) when isIntegerType (T)

RULE: ~ tv (I , T) ⇀ arithInterpret (T , $\sim_{Int}$ I) when isIntegerType (T)

RULE: tv (I_1 , T) ^ tv (I_2 , T) ⇀ arithInterpret (T , I_1 xor_{Int} I_2) when isIntegerType (T)

END MODULE

MODULE COMMON-SEMANTICS-ARITHMETIC
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE
K ::= waitingForOffset(*Nat* , *Type*)
| computePointerDifference(*Int* , *Int* , K) [strict(3)]

K RULES:

RULE: $\langle \underbrace{\text{tv (} \mathit{Offset} \text{ , } T' \text{)} \curvearrowright \text{waitingForOffset (} \mathit{Base} \text{ , } T \text{)}}_{\text{tv (loc (} \mathit{Base} \text{ , } \mathit{Offset} \text{) , } T \text{)}} \cdots \rangle_k$ when $T' =_{Bool}$ cfg:sizeut

RULE: $\langle \underbrace{\text{tv (loc (} \mathit{Block} \text{ , } \mathit{Offset} \text{) , pointerType (} T' \text{)) + tv (} I_2 \text{ , } T \text{)}}_{\text{tv (} \mathit{Offset} \text{ , cfg:sizeut) + sizeofType (} T' \text{) * tv (} I_2 \text{ , cfg:sizeut) } \curvearrowright \text{ waitingForOffset (} \mathit{Block} \text{ , pointerType (} T' \text{))} \cdots \rangle_k$ when isIntegerType (T) $\wedge_{Bool}$ $T' \neq_{Bool}$ void

RULE: $\langle \underbrace{\text{tv (loc (} \mathit{Block} \text{ , } \mathit{Offset} \text{) , pointerType (} T' \text{)) - tv (} I_2 \text{ , } T \text{)}}_{\text{tv (} \mathit{Offset} \text{ , cfg:sizeut) - sizeofType (} T' \text{) * tv (} I_2 \text{ , cfg:sizeut) } \curvearrowright \text{ waitingForOffset (} \mathit{Block} \text{ , pointerType (} T' \text{))} \cdots \rangle_k$ when isIntegerType (T) $\wedge_{Bool}$ $T' \neq_{Bool}$ void

START-POINTER-DIFFERENCE RULE: tv (I_1 , pointerType (T)) - tv (I_2 , pointerType (T)) $\rightharpoonup$ computePointerDifference (I_1 , I_2 , sizeofType (T))

POINTER-DIFFERENCE RULE: computePointerDifference (loc (Block , Offset_1) , loc (Block , Offset_2) , tv (Size , —)) $\rightharpoonup$ tv ($\mathit{Offset}_1 -_{Int} \mathit{Offset}_2 \div_{Int} \mathit{Size}$, cfg:ptrdiffut) when $\mathit{Offset}_1 -_{Int} \mathit{Offset}_2 \%_{Int} \mathit{Size} =_{Bool} 0$

RULE: tv (I_1 , T) + tv (I_2 , T) $\rightharpoonup$ arithInterpret (T , $I_1 +_{Rat} I_2$) when isIntegerType (T)

RULE: tv (I_1 , T) - tv (I_2 , T) $\rightharpoonup$ arithInterpret (T , $I_1 -_{Int} I_2$) when isIntegerType (T)

RULE: tv (I_1 , T) * tv (I_2 , T) $\rightharpoonup$ arithInterpret (T , $I_1 *_{Int} I_2$) when isIntegerType (T)

RULE: tv (I_1 , T) / tv (I_2 , T) $\rightharpoonup$ arithInterpret (T , $I_1 \div_{Int} I_2$) when isIntegerType (T)

RULE: tv (I_1 , T) % tv (I_2 , T) $\rightharpoonup$ arithInterpret (T , $I_1 \%_{Int} I_2$) when isIntegerType (T)

RULE: - tv (I_1 , T) $\rightharpoonup$ arithInterpret (T , $-_{Int} I_1$) when isIntegerType (T)

RULE: tv (F_1 , T) + tv (F_2 , T) $\rightharpoonup$ arithInterpret (T , $F_1 +_{Float} F_2$)

RULE: tv (F_1 , T) - tv (F_2 , T) $\rightharpoonup$ arithInterpret (T , $F_1 -_{Float} F_2$)

RULE: tv (F_1 , T) * tv (F_2 , T) $\rightharpoonup$ arithInterpret (T , $F_1 *_{Float} F_2$)

RULE: tv (F_1 , T) / tv (F_2 , T) $\rightharpoonup$ arithInterpret (T , $F_1 /_{Float} F_2$)

RULE: - tv (F , T) $\rightharpoonup$ arithInterpret (T , -Float F)

RULE: tv (F_1 , T) < tv (F_2 , T) $\rightharpoonup$ if F_1 ensuremath<$_{Float}$ F_2 then tv (1 , int) else tv (0 , int) fi

RULE: tv (F_1 , T) <= tv (F_2 , T) $\rightharpoonup$ if $F_1 \leq_{Float} F_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (F_1 , T) > tv (F_2 , T) $\rightharpoonup$ if $F_1 >_{Float} F_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (F_1 , T) >= tv (F_2 , T) $\rightharpoonup$ if $F_1 \geq_{Float} F_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (F_1 , T) == tv (F_2 , T) ⇀ if $F_1 =_{Bool} F_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (F_1 , T) != tv (F_2 , T) ⇀ if $F_1 \neq_{Bool} F_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (I_1 , T) < tv (I_2 , T) ⇀ if $I_1 <_{Int} I_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (I_1 , T) <= tv (I_2 , T) ⇀ if $I_1 \leq_{Int} I_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (I_1 , T) > tv (I_2 , T) ⇀ if $I_1 >_{Int} I_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (I_1 , T) >= tv (I_2 , T) ⇀ if $I_1 \geq_{Int} I_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (I_1 , T) == tv (I_2 , T) ⇀ if $I_1 =_{Bool} I_2$ then tv (1 , int) else tv (0 , int) fi

RULE: tv (I_1 , T) != tv (I_2 , T) ⇀ if $I_1 \neq_{Bool} I_2$ then tv (1 , int) else tv (0 , int) fi

RULE: ! tv (V , —) ⇀ tv (0 , int) when $\neg_{Bool} V =_{Bool} 0 \vee_{Bool} V =_{Bool} 0.0 \vee_{Bool} V =_{Bool}$ loc (0 , 0)

RULE: ! tv (V , —) ⇀ tv (1 , int) when $V =_{Bool} 0 \vee_{Bool} V =_{Bool} 0.0 \vee_{Bool} V =_{Bool}$ loc (0 , 0)

END MODULE

MODULE COMMON-SEMANTICS-MEMBERS
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE
K ::= dot-aux(K , Id , K) [strict(3)]

K RULES:

ARROW-STRUCT RULE: $\langle \underset{\text{* tv (} Base \text{ , pointerType (structType (} S \text{))) . } X}{\text{tv (} Base \text{ , pointerType (structType (} S \text{))) -> } X} \cdots \rangle_k$

ARROW-UNION RULE: $\langle \underset{\text{* tv (} Base \text{ , pointerType (unionType (} S \text{))) . } X}{\text{tv (} Base \text{ , pointerType (unionType (} S \text{))) -> } X} \cdots \rangle_k$

RULE: $\langle \underset{\text{dot-aux (} E \text{ , } F \text{ , typeof (} E \text{ . } F \text{))}}{E \text{ . } F} \cdots \rangle_k$

RULE: $\langle \underset{\text{\& } E \text{ . } F}{\text{dot-aux (} E \text{ , } F \text{ , arrayType (— , —))}} \cdots \rangle_k$

RULE: $\langle \underset{E \curvearrowright \text{ dot-aux (HOLE , } F \text{ , } T \text{)}}{\text{dot-aux (} E \text{ , } F \text{ , } T \text{)}} \cdots \rangle_k$

RULE: $\langle \underset{E \curvearrowright \text{ dot-aux (HOLE , } F \text{ , pointerType (} T \text{)) }}{\text{dot-aux (} E \text{ , } F \text{ , pointerType (} T \text{))}} \cdots \rangle_k$

RULE: $\langle \underset{E \curvearrowright \text{ dot-aux (HOLE , } F \text{ , structType (} X \text{)) }}{\text{dot-aux (} E \text{ , } F \text{ , structType (} X \text{))}} \cdots \rangle_k$

RULE: $\langle \underset{E \curvearrowright \text{ dot-aux (HOLE , } F \text{ , unionType (} X \text{)) }}{\text{dot-aux (} E \text{ , } F \text{ , unionType (} X \text{))}} \cdots \rangle_k$

RULE: $\langle \underset{E \curvearrowright \text{ dot-aux (HOLE , } F \text{ , bitfieldType (} T \text{ , } N \text{)) }}{\text{dot-aux (} E \text{ , } F \text{ , bitfieldType (} T \text{ , } N \text{))}} \cdots \rangle_k$

RULE: $\langle \underset{\text{extractField (} L \text{ , } T \text{ , } F \text{)}}{\text{atv (} L \text{ , } T \text{) } \curvearrowright \text{ dot-aux (HOLE , } F \text{ , —)}} \cdots \rangle_k$

END MODULE

MODULE COMMON-SEMANTICS-DEREFERENCE
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE

K RULES:

RULE: $\langle \underline{\texttt{*}\ \texttt{tv}\ (\ \mathit{Loc}\ ,\ \texttt{pointerType}\ (\ \texttt{arrayType}\ (\ T\ ,\ N\ )\ )\ )} \cdots \rangle_{\texttt{k}}$
$\texttt{tv}\ (\ \mathit{Loc}\ ,\ \texttt{pointerType}\ (\ T\ )\ )$

LOOKUP-FUNCTION RULE: $\langle \underline{\texttt{*}\ \texttt{tv}\ (\ \texttt{loc}\ (\ \mathit{Block}\ ,\ \mathit{Offset}\ )\ ,\ \texttt{pointerType}\ (\ \texttt{functionType}\ (\ T\ ,\ L\ )\ )\ )} \cdots \rangle_{\texttt{k}}\ \langle \cdots\ \mathit{Block} \mapsto \texttt{memblock}\ (\ \text{—}\ ,\ \text{—}\ \ \mathit{Offset} \mapsto V\ )\ \cdots \rangle_{\texttt{mem}}$
V

DEREF RULE: $\langle \underline{\texttt{*}\ \texttt{tv}\ (\ \mathit{Loc}\ ,\ \texttt{pointerType}\ (\ T\ )\ )} \cdots \rangle_{\texttt{k}}$
$\texttt{readFromMem}\ (\ \mathit{Loc}\ ,\ T\ )$

DEREF-POINTER RULE: $\langle \underline{\texttt{*}\ \texttt{tv}\ (\ \mathit{Loc}\ ,\ \texttt{pointerType}\ (\ \texttt{pointerType}\ (\ T\ )\ )\ )} \cdots \rangle_{\texttt{k}}$
$\texttt{readFromMem}\ (\ \mathit{Loc}\ ,\ \texttt{pointerType}\ (\ T\ )\ )$

DEREF-STRUCT RULE: $\langle \underline{\texttt{*}\ \texttt{tv}\ (\ \mathit{Loc}\ ,\ \texttt{pointerType}\ (\ \texttt{structType}\ (\ X\ )\ )\ )} \cdots \rangle_{\texttt{k}}$
$\texttt{readFromMem}\ (\ \mathit{Loc}\ ,\ \texttt{structType}\ (\ X\ )\ )$

DEREF-UNION RULE: $\langle \underline{\texttt{*}\ \texttt{tv}\ (\ \mathit{Loc}\ ,\ \texttt{pointerType}\ (\ \texttt{unionType}\ (\ X\ )\ )\ )} \cdots \rangle_{\texttt{k}}$
$\texttt{readFromMem}\ (\ \mathit{Loc}\ ,\ \texttt{unionType}\ (\ X\ )\ )$

END MODULE

MODULE COMMON-SEMANTICS-REFERENCE
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE
Type ::= fixPointerType(*Type*)

K RULES:
CONTEXT: & □ -> X
RULE: & * E ⇀ E

RULE: & E_1 [E_2] ⇀ E_1 + E_2

REF-REGISTER RULE: ⟨ $\frac{\text{\& } X}{\text{registerLocation (} X \text{)}}$ ⋯⟩$_{\text{k}}$ ⟨⋯ $X \mapsto$ — ⋯⟩$_{\text{registers}}$

REF RULE: ⟨ $\frac{\text{\& } X}{\text{tv (} Loc \text{ , pointerType (} T \text{))}}$ ⋯⟩$_{\text{k}}$ ⟨⋯ $X \mapsto Loc$ ⋯⟩$_{\text{env}}$ ⟨⋯ $X \mapsto T$ ⋯⟩$_{\text{types}}$

ADDRESS-OF-ARROW-STRUCT RULE: ⟨ $\frac{\text{\& tv (} Base \text{ , pointerType (structType (} S \text{))) -> } X}{\text{cast (fixPointerType (} T \text{) , figureOffset (} Base \text{ , calcStructSize-aux (} L_1 \text{ , 0) , } T \text{))}}$ ⋯⟩$_{\text{k}}$ ⟨⋯ `struct` (S) ↦ L_1 , typedField (T , X) , — ⋯⟩$_{\text{structs}}$

ADDRESS-OF-ARROW-UNION RULE: ⟨ $\frac{\text{\& tv (} Base \text{ , pointerType (unionType (} S \text{))) -> } X}{\text{cast (fixPointerType (} T \text{) , tv (} Base \text{ , pointerType (void)))}}$ ⋯⟩$_{\text{k}}$ ⟨⋯ `union` (S) ↦ — , typedField (T , X) , — ⋯⟩$_{\text{structs}}$

ADDRESS-OF-DOT RULE: ⟨ $\frac{\text{\& } E \text{ . } X}{\text{\& \& } E \text{ -> } X}$ ⋯⟩$_{\text{k}}$

MACRO: fixPointerType (T) = pointerType (T)
MACRO: fixPointerType (arrayType (T , —)) = pointerType (T)
MACRO: fixPointerType (structType (X)) = pointerType (structType (X))
MACRO: fixPointerType (unionType (X)) = pointerType (unionType (X))
MACRO: fixPointerType (pointerType (T)) = pointerType (pointerType (T))
MACRO: fixPointerType (bitfieldType (T , N)) = pointerType (bitfieldType (T , N))
END MODULE

MODULE COMMON-SEMANTICS-INCREMENT-AND-DECREMENT
IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE

K ::= postOpRef(K , *KLabel*)
| postInc(K , K , *Type*) [strict(2)]
| postDec(K , K , *Type*) [strict(2)]

K RULES:

CONTEXT: postOpRef (* □ , —)

MAKE-POSTINC-REF RULE: $\langle \frac{E\ ++}{\text{postOpRef (* \& } E \text{ , \_++)}} \cdots \rangle_k$

MAKE-POSTDEC-REF RULE: $\langle \frac{E\ --}{\text{postOpRef (* \& } E \text{ , \_--)}} \cdots \rangle_k$

POST-INCREMENT-START RULE: $\langle \frac{\text{postOpRef (* tv (} Loc \text{ , pointerType (} T \text{)) , \_++)}}{\text{postInc (} Loc \text{ , readFromMem (} Loc \text{ , } T \text{) , } T \text{)}} \cdots \rangle_k$

POST-INCREMENT RULE: $\langle \frac{\text{postInc (} Loc \text{ , tv (} I \text{ , } T \text{) , } T \text{)}}{\text{assign (* tv (} Loc \text{ , pointerType (} T \text{)) , tv (} I \text{ , } T \text{) + tv (1 , } T \text{)) } \curvearrowright \text{ discard } \curvearrowright \text{ tv (} I \text{ , } T \text{)}} \cdots \rangle_k$ when isIntegerType (T)

POST-INCREMENT-PROMOTE RULE: $\langle \frac{\text{postInc (} Loc \text{ , tv (} I \text{ , pointerType (} T \text{)) , pointerType (} T \text{))}}{\text{assign (* tv (} Loc \text{ , pointerType (pointerType (} T \text{))) , tv (} I \text{ , pointerType (} T \text{)) + tv (1 , int)) } \curvearrowright \text{ discard } \curvearrowright \text{ tv (} I \text{ , pointerType (} T \text{))}} \cdots \rangle_k$

POST-DECREMENT-START RULE: $\langle \frac{\text{postOpRef (* tv (} Loc \text{ , pointerType (} T \text{)) , \_--)}}{\text{postDec (} Loc \text{ , readFromMem (} Loc \text{ , } T \text{) , } T \text{)}} \cdots \rangle_k$

POST-DECREMENT RULE: $\langle \frac{\text{postDec (} Loc \text{ , tv (} I \text{ , } T \text{) , } T \text{)}}{\text{assign (* tv (} Loc \text{ , pointerType (} T \text{)) , tv (} I \text{ , } T \text{) - tv (1 , } T \text{)) } \curvearrowright \text{ discard } \curvearrowright \text{ tv (} I \text{ , } T \text{)}} \cdots \rangle_k$ when isIntegerType (T)

POST-DECREMENT-PROMOTE RULE: $\langle \frac{\text{postDec (} Loc \text{ , tv (} I \text{ , pointerType (} T \text{)) , pointerType (} T \text{))}}{\text{assign (* tv (} Loc \text{ , pointerType (pointerType (} T \text{))) , tv (} I \text{ , pointerType (} T \text{)) - tv (1 , int)) } \curvearrowright \text{ discard } \curvearrowright \text{ tv (} I \text{ , pointerType (} T \text{))}} \cdots \rangle_k$

END MODULE

```
MODULE COMMON-C-EXPRESSIONS
  IMPORTS COMMON-SEMANTICS-SIZEOF
  IMPORTS COMMON-SEMANTICS-IDENTIFIERS
  IMPORTS COMMON-SEMANTICS-FUNCTION-CALLS
  IMPORTS COMMON-SEMANTICS-ARRAY-SUBSCRIPTING
  IMPORTS COMMON-SEMANTICS-ASSIGNMENT
  IMPORTS COMMON-SEMANTICS-LITERALS
  IMPORTS COMMON-SEMANTICS-BITWISE
  IMPORTS COMMON-SEMANTICS-ARITHMETIC
  IMPORTS COMMON-SEMANTICS-MEMBERS
  IMPORTS COMMON-SEMANTICS-DEREFERENCE
  IMPORTS COMMON-SEMANTICS-REFERENCE
  IMPORTS COMMON-SEMANTICS-INCREMENT-AND-DECREMENT
END MODULE
```

```
MODULE COMMON-STATEMENTS-INCLUDE
  IMPORTS COMMON-INCLUDE
  K ::= preparedWhile( K , K )
END MODULE
```

MODULE COMMON-SEMANTICS-IF-THEN
IMPORTS COMMON-STATEMENTS-INCLUDE

K RULES:

IF-THEN-TRUE RULE: $\langle \underbrace{\mathsf{if(\ tv\ (\ } V \mathsf{\ ,\ —\ )\ )}\ S}_{\mathsf{sequencePoint} \curvearrowright S}\ \cdots\rangle_{\mathsf{k}}$ when $\neg_{Bool}\ V =_{Bool} \mathsf{0} \vee_{Bool} V =_{Bool} \mathsf{0.0} \vee_{Bool} V =_{Bool} \mathsf{loc\ (\ 0\ ,\ 0\ )}$

IF-THEN-FALSE RULE: $\langle \underbrace{\mathsf{if(\ tv\ (\ } V \mathsf{\ ,\ —\ )\ )}\ S}_{\mathsf{sequencePoint}}\ \cdots\rangle_{\mathsf{k}}$ when $V =_{Bool} \mathsf{0} \vee_{Bool} V =_{Bool} \mathsf{0.0} \vee_{Bool} V =_{Bool} \mathsf{loc\ (\ 0\ ,\ 0\ )}$

IF-THEN-ELSE-TRUE RULE: $\langle \underbrace{\mathsf{if(\ tv\ (\ } V \mathsf{\ ,\ —\ )\ )}\ S \mathsf{\ else\ } S'}_{\mathsf{sequencePoint} \curvearrowright S}\ \cdots\rangle_{\mathsf{k}}$ when $\neg_{Bool}\ V =_{Bool} \mathsf{0} \vee_{Bool} V =_{Bool} \mathsf{0.0} \vee_{Bool} V =_{Bool} \mathsf{loc\ (\ 0\ ,\ 0\ )}$

IF-THEN-ELSE-FALSE RULE: $\langle \underbrace{\mathsf{if(\ tv\ (\ } V \mathsf{\ ,\ —\ )\ )}\ S \mathsf{\ else\ } S'}_{\mathsf{sequencePoint} \curvearrowright S'}\ \cdots\rangle_{\mathsf{k}}$ when $V =_{Bool} \mathsf{0} \vee_{Bool} V =_{Bool} \mathsf{0.0} \vee_{Bool} V =_{Bool} \mathsf{loc\ (\ 0\ ,\ 0\ )}$

END MODULE

MODULE COMMON-SEMANTICS-WHILE
IMPORTS COMMON-STATEMENTS-INCLUDE

K RULES:

WHILE-MARK RULE: $\langle \underset{\text{preparedWhile}\,(\,B\,,\,S\,)\,\curvearrowright\,\texttt{break}}{\underbrace{\text{while}(\,B\,)\;S\,\curvearrowright\,K}} \rangle_{\text{k}}\;\langle\; \underset{K}{\underbrace{\cdot}}\;\cdots\rangle_{\text{loopStack}}$

WHILE RULE: $\langle \underset{\text{if}(\,B\,)\;S\,\curvearrowright\,\text{preparedWhile}\,(\,B\,,\,S\,)}{\underbrace{\text{preparedWhile}\,(\,B\,,\,S\,)}}\;\cdots\rangle_{\text{k}}$

END MODULE

MODULE COMMON-SEMANTICS-SWITCH
IMPORTS COMMON-STATEMENTS-INCLUDE

K RULES:

RULE: $\langle \underline{\mathsf{switch}(\ SN\ )(\ \mathsf{tv}\ (\ V\ ,\ \text{—}\ )\ )\ K_1 \curvearrowright \text{—}} \rangle_{\mathsf{k}}\ \langle F \rangle_{\mathsf{currentFunction}}\ \langle \underline{\text{—}} \rangle_{\mathsf{loopStack}}\ \langle \cdots\ \mathsf{kpair}\ (\ F\ ,\ \mathsf{case}\ (\ SN\ ,\ V\ )\ ) \mapsto \mathsf{kpair}\ (\ K_2\ ,\ S\ )\ \cdots \rangle_{\mathsf{gotoMap}}$
K_2 S

RULE: $\langle \underline{\mathsf{switch}(\ SN\ )(\ \mathsf{tv}\ (\ V\ ,\ \text{—}\ )\ )\ K_1 \curvearrowright \text{—}} \rangle_{\mathsf{k}}\ \langle F \rangle_{\mathsf{currentFunction}}\ \langle \underline{\text{—}} \rangle_{\mathsf{loopStack}}\ \langle LM\ \ \mathsf{kpair}\ (\ F\ ,\ \mathsf{defaultCase}\ (\ SN\ )\ ) \mapsto \mathsf{kpair}\ (\ K_2\ ,\ S\ ) \rangle_{\mathsf{gotoMap}}$ when $\neg_{Bool}\ \$hasMapping(LM\ ,\ \mathsf{kpair}\ (\ F\ ,\ \mathsf{case}\ (\ SN\ ,\ V\ )\ ))$
K_2 S

RULE: $\langle \underline{\mathsf{switch}(\ SN\ )(\ \mathsf{tv}\ (\ V\ ,\ \text{—}\ )\ )\ K_1} \cdot \cdots \rangle_{\mathsf{k}}\ \langle F \rangle_{\mathsf{currentFunction}}\ \langle LM \rangle_{\mathsf{gotoMap}}$ when $\neg_{Bool}\ \$hasMapping(LM\ ,\ \mathsf{kpair}\ (\ F\ ,\ \mathsf{defaultCase}\ (\ SN\ )\ ))\ \vee_{Bool}\ \$hasMapping(LM\ ,\ \mathsf{kpair}\ (\ F\ ,\ \mathsf{case}\ (\ SN\ ,\ V\ )\ ))$

CASE-FALL-THROUGH RULE: $\langle \underline{\mathsf{case}(\ \text{—}\ )\ \text{—}\ :\ K} \cdot \cdots \rangle_{\mathsf{k}}$
K

DEFAULT-FALL-THROUGH RULE: $\langle \underline{\mathsf{default}(\ \text{—}\ ):\ K} \cdot \cdots \rangle_{\mathsf{k}}$
K

END MODULE

MODULE COMMON-SEMANTICS-GOTO
IMPORTS COMMON-STATEMENTS-INCLUDE
K ::= calculateGotoMap(Id , K , $List$, K)

K RULES:
CONTEXT: calculateGotoMap (— , case(—) □ : — $\curvearrowright$ — , — , —)
GOTO RULE: $\langle \underline{\texttt{goto}\ X \curvearrowright —} \rangle_k$ $\langle F \rangle_{\text{currentFunction}}$ $\langle \underline{—} \rangle_{\text{loopStack}}$ $\langle \cdots$ kpair (F , X) $\mapsto$ kpair (K , S) $\cdots \rangle_{\text{gotoMap}}$

RULE: $\langle$ calculateGotoMap (— , · , — , —) $\cdots \rangle_k$
.

RULE: $\langle$ calculateGotoMap (X , L ($Args$) $\curvearrowright$ K , S , $Tail$) $\cdots \rangle_k$ when $L =_{Bool}$ `Local` $\vee_{Bool}$ $L =_{Bool}$ _; $\vee_{Bool}$ $L =_{Bool}$ `break` $\vee_{Bool}$ $L =_{Bool}$ `goto_` $\vee_{Bool}$ $L =_{Bool}$ `return_` $\vee_{Bool}$ $L =_{Bool}$ `return`
calculateGotoMap (X , K , S , $Tail$)

RULE: $\langle$ calculateGotoMap (X , `Block` ($\cdot List\{K\}$) $\curvearrowright$ K , S , $Tail$) $\cdots \rangle_k$
calculateGotoMap (X , K , S , $Tail$)

RULE: $\langle$ calculateGotoMap (X , `Block` (Arg) $\curvearrowright$ K , S , $Tail$) $\cdots \rangle_k$
calculateGotoMap (X , $Arg \curvearrowright K$, S , $Tail$)

RULE: $\langle$ calculateGotoMap (X , L (— , Arg) $\curvearrowright$ K , S , $Tail$) $\cdots \rangle_k$ when $L =_{Bool}$ if(_)_
calculateGotoMap (X , $Arg \curvearrowright K$, S , $Tail$)

RULE: $\langle$ calculateGotoMap (X , if(—) Arg_1 else $Arg_2 \curvearrowright K$, S , $Tail$) $\cdots \rangle_k$
calculateGotoMap (X , Arg_1 , S , $Tail \curvearrowright K$) $\curvearrowright$ calculateGotoMap (X , Arg_2 , S , $Tail \curvearrowright K$) $\curvearrowright$ calculateGotoMap (X , K , S , $Tail$)

RULE: $\langle$ calculateGotoMap (X , $Target$: $Arg \curvearrowright K$, S , $Tail$) $\cdots \rangle_k$ $\langle \cdots$ kpair (X , $Target$) $\mapsto$ kpair ($Arg \curvearrowright K \curvearrowright Tail$, S) $\cdots \rangle_{\text{gotoMap}}$
calculateGotoMap (X , $Arg \curvearrowright K$, S , $Tail$)
.

RULE: $\langle$ calculateGotoMap (X , case(SN) tv ($Target$, T) : $Arg \curvearrowright K$, S , $Tail$) $\cdots \rangle_k$ $\langle \cdots$ kpair (X , case (SN , $Target$)) $\mapsto$ kpair ($Arg \curvearrowright K \curvearrowright Tail$, S) $\cdots \rangle_{\text{gotoMap}}$
calculateGotoMap (X , $Arg \curvearrowright K$, S , $Tail$)
.

RULE: $\langle$ calculateGotoMap (X , default(SN): $Arg \curvearrowright K$, S , $Tail$) $\cdots \rangle_k$ $\langle \cdots$ kpair (X , defaultCase (SN)) $\mapsto$ kpair ($Arg \curvearrowright K \curvearrowright Tail$, S) $\cdots \rangle_{\text{gotoMap}}$
calculateGotoMap (X , $Arg \curvearrowright K$, S , $Tail$)
.

RULE: $\langle$ calculateGotoMap (X , while(B) $S \curvearrowright K$, S' , $Tail$) $\cdots \rangle_k$
calculateGotoMap (X , $S \curvearrowright$ preparedWhile (B , S) $\curvearrowright$ `break` , $K \curvearrowright Tail$ S' , $Tail$)

RULE: $\langle$ calculateGotoMap (X , preparedWhile (B , S) $\curvearrowright$ `break` $\curvearrowright$ — , K S' , $Tail$) $\cdots \rangle_k$
calculateGotoMap (X , K , S' , $Tail$)

RULE: $\langle$ calculateGotoMap (X , switch(—)(B) $S \curvearrowright K$, S , $Tail$) $\cdots \rangle_k$
calculateGotoMap (X , $S \curvearrowright$ `break` $\curvearrowright$ popLoop , $K \curvearrowright Tail$ S , $Tail$)

RULE: $\langle$ calculateGotoMap (X , popLoop , K S , $Tail$) $\cdots \rangle_k$
calculateGotoMap (X , K , S , $Tail$)

MACRO: calculateGotoMap (X , K) = calculateGotoMap (X , K , · , ·)
END MODULE

MODULE COMMON-SEMANTICS-RETURN
IMPORTS COMMON-STATEMENTS-INCLUDE

K RULES:

RETURN-VALUE-CLEAN-LOCAL RULE: $\langle \texttt{return}\ V\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ \underset{\cdot}{\mathsf{loc}\ (\ \mathit{Block}\ ,\ \_\ )}\ \cdots \rangle_{\mathsf{locals}}\ \langle \cdot \rangle_{\mathsf{buffer}}\ \langle \cdots\ \underset{\cdot}{\mathit{Block} \mapsto \_}\ \cdots \rangle_{\mathsf{mem}}$

RETURN-CLEAN-LOCAL RULE: $\langle \texttt{return}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ \underset{\cdot}{\mathsf{loc}\ (\ \mathit{Block}\ ,\ \_\ )}\ \cdots \rangle_{\mathsf{locals}}\ \langle \cdot \rangle_{\mathsf{buffer}}\ \langle \cdots\ \underset{\cdot}{\mathit{Block} \mapsto \_}\ \cdots \rangle_{\mathsf{mem}}$

RETURN RULE: $\langle \underset{\mathsf{skipval} \curvearrowright K}{\texttt{return} \curvearrowright \_} \rangle_{\mathsf{k}}\ \langle \underset{E}{\_} \rangle_{\mathsf{env}}\ \langle \underset{L}{\cdot} \rangle_{\mathsf{locals}}\ \langle \underset{LS}{\_} \rangle_{\mathsf{loopStack}}\ \langle \underset{T}{\_} \rangle_{\mathsf{types}}\ \langle \underset{X}{\_} \rangle_{\mathsf{currentFunction}}\ \langle \underset{\cdot}{\mathsf{List}\ (\ \langle \cdots\ \langle X \rangle_{\mathsf{currentFunction}}\ \langle K \rangle_{\mathsf{continuation}}\ \langle L \rangle_{\mathsf{locals}}\ \langle E \rangle_{\mathsf{env}}\ \langle LS \rangle_{\mathsf{loopStack}}\ \langle T \rangle_{\mathsf{types}} \rangle_{\mathsf{stackFrame}}\ )} \cdots \rangle_{\mathsf{callStack}}$

RETURN-VALUE RULE: $\langle \underset{\mathsf{sequencePoint} \curvearrowright V \curvearrowright K}{\texttt{return}\ V \curvearrowright \_} \rangle_{\mathsf{k}}\ \langle \underset{E}{\_} \rangle_{\mathsf{env}}\ \langle \underset{L}{\cdot} \rangle_{\mathsf{locals}}\ \langle \underset{LS}{LS'} \rangle_{\mathsf{loopStack}}\ \langle \underset{T}{\_} \rangle_{\mathsf{types}}\ \langle \underset{X}{\_} \rangle_{\mathsf{currentFunction}}\ \langle \underset{\cdot}{\mathsf{List}\ (\ \langle \cdots\ \langle X \rangle_{\mathsf{currentFunction}}\ \langle K \rangle_{\mathsf{continuation}}\ \langle L \rangle_{\mathsf{locals}}\ \langle E \rangle_{\mathsf{env}}\ \langle LS \rangle_{\mathsf{loopStack}}\ \langle T \rangle_{\mathsf{types}} \rangle_{\mathsf{stackFrame}}\ )} \cdots \rangle_{\mathsf{callStack}}$

END MODULE

MODULE COMMON-C-STATEMENTS
IMPORTS COMMON-STATEMENTS-INCLUDE
IMPORTS COMMON-SEMANTICS-IF-THEN
IMPORTS COMMON-SEMANTICS-WHILE
IMPORTS COMMON-SEMANTICS-SWITCH
IMPORTS COMMON-SEMANTICS-GOTO
IMPORTS COMMON-SEMANTICS-RETURN

K RULES:

RULE: EmptyStatement‘; ⇀ ·

RULE: Block() ⇀ ·

RULE: $S_1 \;\; S_2 \rightharpoonup S_1 \curvearrowright S_2$

SKIP-LABEL RULE: $\langle \underline{L : S} \;\cdots\rangle_{\mathsf{k}}$ / S

DISSOLVE-BLOCK RULE: $\langle \underline{\texttt{Block}\;(\;B\;)} \;\cdots\rangle_{\mathsf{k}}$ / B

VALUE-STATEMENT RULE: $\langle \underline{\quad V\;; \quad} \;\cdots\rangle_{\mathsf{k}}$ / sequencePoint

BREAK RULE: $\langle \underline{\texttt{break} \curvearrowright \_\!\_} \rangle_{\mathsf{k}} \;\langle \underline{K} \;\cdots\rangle_{\mathsf{loopStack}}$ / K ; $\cdot$

END MODULE

MODULE COMMON-C-STANDARD-LIBRARY

IMPORTS COMMON-INCLUDE

Value ::= builtin(*Id*)

K ::= BagK(*Bag*)
| printf-aux(*Nat* , *Nat* , *List{KResult}*)
| printf-string(*Nat* , *Nat*)
| printf-%(*Nat* , *Nat* , *List{KResult}*)
| addPrintfString

Int ::= zeroToOne(*Int*)

K RULES:

DEBUG-IS-VALUE RULE: $\langle \underset{\text{builtin (debug)}}{\text{debug}} \cdots \rangle_{\text{k}}$

DEBUG RULE: $\langle \underset{\text{skipval}}{\text{application (builtin (debug) , —)}} \cdots \rangle_{\text{k}}$

SETJMP-IS-VALUE RULE: $\langle \underset{\text{builtin (setjmp)}}{\text{setjmp}} \cdots \rangle_{\text{k}}$

SETJMP RULE:
$$\left(\begin{array}{l} \langle B \ \langle \underset{\text{tv (0 , int)}}{\text{application (builtin (setjmp) , tv (loc (} \mathit{Block} \text{ , } \mathit{Offset} \text{) , pointerType (structType (—))))}} \curvearrowright K \rangle_{\text{k}} \rangle_{\text{control}} \ \langle \mathit{CallStack} \rangle_{\text{callStack}} \\ \qquad \langle \cdots \ \mathit{Block} \mapsto \text{memblock (— , } \underset{M \text{ [BagK (} \langle B \ \langle K \rangle_{\text{continuation}} \rangle_{\text{control}} \ \langle \mathit{CallStack} \rangle_{\text{callStack}} \text{) / } \mathit{Offset} \text{]}}{M} \text{) } \cdots \rangle_{\text{mem}} \end{array} \right)$$

LONGJMP-IS-VALUE RULE: $\langle \underset{\text{builtin (longjmp)}}{\text{longjmp}} \cdots \rangle_{\text{k}}$

LONGJMP RULE: $\langle \underset{B \ \langle \text{tv (zeroToOne (} I \text{) , int) } \curvearrowright K \rangle_{\text{k}}}{\langle \text{— } \langle \text{application (builtin (longjmp) , tv (loc (} \mathit{Block} \text{ , } \mathit{Offset} \text{) , pointerType (structType (—))) , tv (} I \text{ , int)) } \curvearrowright \text{—} \rangle_{\text{k}} \rangle_{\text{control}}} \ \langle \underset{\mathit{CallStack}}{\text{—}} \rangle_{\text{callStack}}$
$\langle \cdots \ \mathit{Block} \mapsto \text{memblock (— , — } \mathit{Offset} \mapsto \underset{\text{piece (unknown (} \mathit{Fresh} \text{) , 8)}}{\text{BagK (} \langle B \ \langle K \rangle_{\text{continuation}} \rangle_{\text{control}} \ \langle \mathit{CallStack} \rangle_{\text{callStack}} \text{))}} \ \cdots \rangle_{\text{mem}} \ \langle \underset{\text{s}_{Nat} \ \mathit{Fresh}}{\mathit{Fresh}} \rangle_{\text{freshNat}}$

EXIT-IS-VALUE RULE: $\langle \underset{\text{builtin (exit)}}{\text{exit}} \cdots \rangle_{\text{k}}$

EXIT RULE: $\langle \underset{\text{tv (} I \text{ , int)}}{\text{application (builtin (exit) , tv (} I \text{ , int)) } \curvearrowright \text{—}} \rangle_{\text{k}}$

SQRT-IS-VALUE RULE: $\langle \underset{\text{builtin (sqrt)}}{\text{sqrt}} \cdots \rangle_{\text{k}}$

SQRT RULE: $\langle \underset{\text{tv (sqrtFloat (} F \text{) , double)}}{\text{application (builtin (sqrt) , tv (} F \text{ , double))}} \cdots \rangle_{\text{k}}$

LOG-IS-VALUE RULE: $\langle \underset{\text{builtin (log)}}{\text{log}} \cdots \rangle_{\text{k}}$

LOG RULE: $\langle \underset{\text{tv (logFloat (} F \text{) , double)}}{\text{application (builtin (log) , tv (} F \text{ , double))}} \cdots \rangle_{\text{k}}$

EXP-IS-VALUE RULE: $\langle \underline{\texttt{exp}} \cdots \rangle_k$
$\texttt{builtin (exp)}$

EXP RULE: $\langle \underline{\texttt{application (builtin (exp) , tv (} F \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (expFloat (} F \texttt{) , double)}$

ATAN-IS-VALUE RULE: $\langle \underline{\texttt{atan}} \cdots \rangle_k$
$\texttt{builtin (atan)}$

ATAN RULE: $\langle \underline{\texttt{application (builtin (atan) , tv (} F \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (atanFloat (} F \texttt{) , double)}$

ASIN-IS-VALUE RULE: $\langle \underline{\texttt{asin}} \cdots \rangle_k$
$\texttt{builtin (asin)}$

ASIN RULE: $\langle \underline{\texttt{application (builtin (asin) , tv (} F \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (asinFloat (} F \texttt{) , double)}$

ATAN2-IS-VALUE RULE: $\langle \underline{\texttt{atan2}} \cdots \rangle_k$
$\texttt{builtin (atan2)}$

ATAN2 RULE: $\langle \underline{\texttt{application (builtin (atan2) , tv (} F \texttt{ , double) , tv (} F' \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (atanFloat (} F \texttt{ , } F' \texttt{) , double)}$

TAN-IS-VALUE RULE: $\langle \underline{\texttt{tan}} \cdots \rangle_k$
$\texttt{builtin (tan)}$

TAN RULE: $\langle \underline{\texttt{application (builtin (tan) , tv (} F \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (tanFloat (} F \texttt{) , double)}$

FLOOR-IS-VALUE RULE: $\langle \underline{\texttt{floor}} \cdots \rangle_k$
$\texttt{builtin (floor)}$

COS-IS-VALUE RULE: $\langle \underline{\texttt{cos}} \cdots \rangle_k$
$\texttt{builtin (cos)}$

COS RULE: $\langle \underline{\texttt{application (builtin (cos) , tv (} F \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (cosFloat (} F \texttt{) , double)}$

FMOD-IS-VALUE RULE: $\langle \underline{\texttt{fmod}} \cdots \rangle_k$
$\texttt{builtin (fmod)}$

FMOD RULE: $\langle \underline{\texttt{application (builtin (fmod) , tv (} F \texttt{ , double) , tv (} F' \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (} F \ \%_{Float} \ F' \texttt{ , double)}$

SIN-IS-VALUE RULE: $\langle \underline{\texttt{sin}} \cdots \rangle_k$
$\texttt{builtin (sin)}$

SIN RULE: $\langle \underline{\texttt{application (builtin (sin) , tv (} F \texttt{ , double))}} \cdots \rangle_k$
$\texttt{tv (sinFloat (} F \texttt{) , double)}$

MALLOC-IS-VALUE RULE: $\langle \frac{\text{malloc}}{\text{builtin (malloc)}} \cdots \rangle_{\text{k}}$

MALLOC RULE: $\langle \frac{\text{application (builtin (malloc) , tv (} N \text{ , } T \text{))}}{\text{alloc (} Loc \text{ , tv (} N \text{ , } T \text{)) } \curvearrowright \text{ tv (} Loc \text{ , pointerType (void))}} \cdots \rangle_{\text{k}} \langle \cdots \frac{\cdot}{Loc \mapsto N} \cdots \rangle_{\text{malloced}} \langle \frac{Loc}{\text{inc (} Loc \text{)}} \rangle_{\text{nextLoc}}$

RAND-IS-VALUE RULE: $\langle \frac{\text{rand}}{\text{builtin (rand)}} \cdots \rangle_{\text{k}}$

RAND RULE: $\langle \frac{\text{application (builtin (rand) , } \cdot List\{K\} \text{)}}{\text{tv (| randomRandom (} Fresh \text{) |}_{Int} \text{ \%}_{Int} \text{ max (int) , int)}} \cdots \rangle_{\text{k}} \langle \frac{Fresh}{\text{s}_{Nat}\ Fresh} \rangle_{\text{freshNat}}$

FREE-IS-VALUE RULE: $\langle \frac{\text{free}}{\text{builtin (free)}} \cdots \rangle_{\text{k}}$

FREE RULE: $\langle \frac{\text{application (builtin (free) , tv (loc (} Block \text{ , } Offset \text{) , pointerType (} T \text{)))}}{\text{skipval}} \cdots \rangle_{\text{k}} \langle \cdots \frac{Block \mapsto \text{memblock (} N \text{ , —)}}{\cdot} \cdots \rangle_{\text{mem}} \langle \cdots \frac{\text{loc (} Block \text{ , } Offset \text{)} \mapsto N}{\cdot} \cdots \rangle_{\text{malloced}}$

CALLOC-IS-VALUE RULE: $\langle \frac{\text{calloc}}{\text{builtin (calloc)}} \cdots \rangle_{\text{k}}$

CALLOC RULE: $\langle \frac{\text{application (builtin (calloc) , tv (} N \text{ , —) , tv (} Size \text{ , —))}}{\text{store } N *_{Nat} Size \text{ New tv (0 , char) atLoc } Loc \curvearrowright \text{ tv (} Loc \text{ , pointerType (void))}} \cdots \rangle_{\text{k}} \langle \cdots \frac{\cdot}{Loc \mapsto N *_{Nat} Size} \cdots \rangle_{\text{malloced}} \langle \frac{Loc}{\text{inc (} Loc \text{)}} \rangle_{\text{nextLoc}}$

PUTCHAR-IS-VALUE RULE: $\langle \frac{\text{putchar}}{\text{builtin (putchar)}} \cdots \rangle_{\text{k}}$

PUTCHAR RULE: $\langle \frac{\text{application (builtin (putchar) , tv (} N \text{ , —))}}{\text{writeToFD (1 , } N \text{) } \curvearrowright \text{ tv (} N \text{ , int)}} \cdots \rangle_{\text{k}}$

LIB-PRINTF-IS-VALUE RULE: $\langle \frac{\text{printf}}{\text{builtin (printf)}} \cdots \rangle_{\text{k}}$

LIB-PRINTF-START RULE: $\langle \frac{\text{application (builtin (printf) , tv (} Loc \text{ , pointerType (—)) , } L \text{)}}{\text{printf-aux (0 , } Loc \text{ , } L \text{)}} \cdots \rangle_{\text{k}}$

LIB-PRINTF-PREPARE-STRING RULE: $\langle \frac{\cdot}{\text{readFromMem (} Loc \text{ , char)}} \curvearrowright \text{printf-string (— , } Loc \text{)} \cdots \rangle_{\text{k}}$

LIB-PRINTF-STRING RULE: $\langle \frac{\text{tv (} N \text{ , —)} \curvearrowright \text{printf-string (} Len \text{ , } Loc \text{)}}{\text{writeToFD (1 , } N \text{)} \curvearrowright \text{printf-string (s}_{Nat}\ Len \text{ , s}_{Nat}\ Loc \text{)}} \cdots \rangle_{\text{k}}$ when $N \neq_{Bool} 0$

LIB-PRINTF-STRING-DONE RULE: $\langle \frac{\text{tv (0 , —)} \curvearrowright \text{printf-string (} Len \text{ , —)}}{\text{tv (} Len \text{ , int)}} \cdots \rangle_{\text{k}}$

LIB-PRINTF-PREPARE-NORMAL RULE: $\langle \frac{\cdot}{\text{readFromMem (} Loc \text{ , char)}} \curvearrowright \text{printf-aux (— , } Loc \text{ , —)} \cdots \rangle_{\text{k}}$

LIB-PRINTF-NORMAL RULE: $\langle \frac{\text{tv (} N \text{ , —)} \curvearrowright \text{printf-aux (} Len \text{ , } Loc \text{ , } L \text{)}}{\text{writeToFD (1 , } N \text{)} \curvearrowright \text{printf-aux (s}_{Nat}\ Len \text{ , s}_{Nat}\ Loc \text{ , } L \text{)}} \cdots \rangle_{\text{k}}$ when $N \leq_{Nat} 255 \wedge_{Bool} N \geq_{Nat} 0 \wedge_{Bool} N \neq_{Bool} 0 \wedge_{Bool} N \neq_{Bool} \text{asciiString (“\%”)}$

LIB-PRINTF-DONE RULE: $\langle \underline{\mathsf{tv}(0, \_) \curvearrowright \mathsf{printf\text{-}aux}(Len, \_, \_)} \cdots \rangle_k$
$\mathsf{tv}(Len, \mathsf{int})$

LIB-PRINTF-NULL RULE: $\langle \underline{\mathsf{printf\text{-}aux}(0, \mathsf{loc}(0, 0), \cdot List\{K\})} \cdots \rangle_k$
$\mathsf{writeToFD}(1, \text{“(null)”}) \curvearrowright \mathsf{tv}(6, \mathsf{int})$

LIB-PRINTF-%-PREPARE RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}aux}(Len, Loc, L)} \cdots \rangle_k$ when $N =_{Bool} \mathsf{asciiString}(\text{“\%”})$
$\mathsf{printf\text{-}\%}(Len, \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-% RULE: $\langle \underline{\cdot} \curvearrowright \mathsf{printf\text{-}\%}(\_, Loc, \_) \cdots \rangle_k$
$\mathsf{readFromMem}(Loc, \mathsf{char})$

LIB-PRINTF-%% RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, L)} \cdots \rangle_k$ when $N =_{Bool} \mathsf{asciiString}(\text{“\%”})$
$\mathsf{writeToFD}(1, N) \curvearrowright \mathsf{printf\text{-}aux}(\mathsf{s}_{Nat}\, Len, \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-0 RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, L)} \cdots \rangle_k$ when $N =_{Bool} \mathsf{asciiString}(\text{“0”})$
$\mathsf{printf\text{-}\%}(Len, \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-WIDTH RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, L)} \cdots \rangle_k$ when $N \leq_{Int} \mathsf{asciiString}(\text{“9”}) \wedge_{Bool} N >_{Int} \mathsf{asciiString}(\text{“0”})$
$\mathsf{printf\text{-}\%}(Len, \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-%X-1 RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, \mathsf{tv}(D, \_), L)} \cdots \rangle_k$ when $N =_{Bool} \mathsf{asciiString}(\text{“p”}) \vee_{Bool} N =_{Bool} \mathsf{asciiString}(\text{“x”}) \wedge_{Bool}$
$\mathsf{lengthString}(\mathsf{Rat2String}(D, 16)) =_{Bool} 1$
$\mathsf{writeToFD}(1, \text{“0”} +_{String} \mathsf{Rat2String}(D, 16)) \curvearrowright \mathsf{printf\text{-}aux}(Len +_{Nat} 2, \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-%X-2 RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, \mathsf{tv}(D, \_), L)} \cdots \rangle_k$ when
$N =_{Bool} \mathsf{asciiString}(\text{“p”}) \vee_{Bool} N =_{Bool} \mathsf{asciiString}(\text{“x”}) \wedge_{Bool} \mathsf{lengthString}(\mathsf{Rat2String}(D, 16)) \neq_{Bool} 1$
$\mathsf{writeToFD}(1, \mathsf{Rat2String}(D, 16)) \curvearrowright \mathsf{printf\text{-}aux}(Len +_{Nat} \mathsf{lengthString}(\mathsf{Rat2String}(D, 16)), \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-%D RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, \mathsf{tv}(D, \_), L)} \cdots \rangle_k$ when
$N =_{Bool} \mathsf{asciiString}(\text{“d”}) \vee_{Bool} N =_{Bool} \mathsf{asciiString}(\text{“u”})$
$\mathsf{writeToFD}(1, \mathsf{Rat2String}(D, 10)) \curvearrowright \mathsf{printf\text{-}aux}(Len +_{Nat} \mathsf{lengthString}(\mathsf{Rat2String}(D, 10)), \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-%C RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, \mathsf{tv}(C, \_), L)} \cdots \rangle_k$ when $N =_{Bool} \mathsf{asciiString}(\text{“c”})$
$\mathsf{writeToFD}(1, C) \curvearrowright \mathsf{printf\text{-}aux}(Len +_{Nat} 1, \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-%F RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, \mathsf{tv}(D, \_), L)} \cdots \rangle_k$ when $N =_{Bool} \mathsf{asciiString}(\text{“f”}) \vee_{Bool}$
$N =_{Bool} \mathsf{asciiString}(\text{“g”})$
$\mathsf{writeToFD}(1, \mathsf{Float2String}(D)) \curvearrowright \mathsf{printf\text{-}aux}(Len +_{Nat} \mathsf{lengthString}(\mathsf{Float2String}(D)), \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-%S RULE: $\langle \underline{\mathsf{tv}(N, \_) \curvearrowright \mathsf{printf\text{-}\%}(Len, Loc, \mathsf{tv}(S, \mathsf{pointerType}(\_)), L)} \cdots \rangle_k$ when $N =_{Bool} \mathsf{asciiString}(\text{“s”})$
$\mathsf{printf\text{-}string}(0, S) \curvearrowright \mathsf{addPrintfString} \curvearrowright \mathsf{printf\text{-}aux}(Len, \mathsf{s}_{Nat}\, Loc, L)$

LIB-PRINTF-%S-DONE RULE: $\langle \underline{\mathsf{tv}(Len', \mathsf{int}) \curvearrowright \mathsf{addPrintfString} \curvearrowright \mathsf{printf\text{-}aux}(Len, Loc, L)} \cdots \rangle_k$
$\mathsf{printf\text{-}aux}(Len +_{Nat} Len', Loc, L)$

MACRO: $\mathsf{debugK} = \mathsf{application}(\mathsf{builtin}(\mathsf{debug}), 0) \curvearrowright \mathsf{discard}$
MACRO: $\mathsf{zeroToOne}(I) = \mathsf{if}\ I =_{Bool} 0\ \mathsf{then}\ 1\ \mathsf{else}\ I\ \mathsf{fi}$
MACRO: $\mathsf{Rat2String}(\mathsf{loc}(N, M), 16) = \text{“loc(”} +_{String} \mathsf{Rat2String}(N, 16) +_{String} \text{“,”} +_{String} \mathsf{Rat2String}(M, 16) +_{String} \text{“)”}$
END MODULE

MODULE COMMON-C-SEMANTICS
IMPORTS COMMON-INCLUDE
IMPORTS COMMON-GLOBAL-DECLARATION
IMPORTS COMMON-LOCAL-DECLARATION
IMPORTS COMMON-C-STANDARD-LIBRARY
IMPORTS COMMON-PARAMETER-BINDING
IMPORTS MEMORY
IMPORTS COMMON-C-CONVERSIONS
IMPORTS COMMON-C-EXPRESSIONS
IMPORTS COMMON-C-STATEMENTS
IMPORTS COMMON-C-TYPING
Bag ::= eval(*Program*)
| eval(*Program* , *List{K}* , *String*)
K ::= extractField-pre(*List{K}* , *Type* , *K* , *K*) [strict(3 4)]
| extractField-aux(*List{K}* , *Type* , *Nat* , *Nat* , *List{K}*)
List{K} ::= explodeToBits(*List{K}*)

K RULES:

RULE: $\langle \overset{\cdot}{\underset{T}{X}} \cdots \rangle_{\text{k}}$ $\langle \cdots$ `typedefName` (X) $\mapsto T$ $\cdots \rangle_{\text{types}}$

RULE: $\langle \overset{\cdot}{\underset{T}{\texttt{struct}\ (\ X\ )}} \cdots \rangle_{\text{k}}$ $\langle \cdots$ `struct` (X) $\mapsto T$ $\cdots \rangle_{\text{types}}$

RULE: $\langle \overset{\cdot}{\underset{T}{\texttt{union}\ (\ X\ )}} \cdots \rangle_{\text{k}}$ $\langle \cdots$ `union` (X) $\mapsto T$ $\cdots \rangle_{\text{types}}$

RULE: $\langle \overset{\cdot}{\underset{T}{\text{enum}\ (\ X\ )}} \cdots \rangle_{\text{k}}$ $\langle \cdots$ enum (X) $\mapsto T$ $\cdots \rangle_{\text{types}}$

RULE: $\langle \overset{\cdot}{\text{sequencePoint}} \cdots \rangle_{\text{k}}$ $\langle \overset{\cdot}{\text{---}} \rangle_{\text{locsWrittenTo}}$

EXTRACTFIELD-FROM-STRUCT-START RULE: $\langle \overset{\cdot}{\text{extractField (} L \text{ , structType (} S \text{) , } F \text{)}} \cdots \rangle_{\text{k}}$
$\overset{\cdot}{\text{extractField-pre (} L \text{ , } T \text{ , figureOffset (loc (0 , 0) , calcStructSize-aux (} L_1 \text{ , 0) , } T \text{) , sizeofType (} T \text{))}}$
$\langle \cdots$ `struct` (S) $\mapsto L_1$, typedField (T , F) , --- $\cdots \rangle_{\text{structs}}$

EXTRACTFIELD-FROM-UNION-START RULE: $\langle \overset{\cdot}{\text{extractField (} L \text{ , unionType (} S \text{) , } F \text{)}} \cdots \rangle_{\text{k}}$ $\langle \cdots$ `union` (S) $\mapsto$ --- , typedField (T , F) , --- $\cdots \rangle_{\text{structs}}$
$\overset{\cdot}{\text{extractField-pre (} L \text{ , } T \text{ , tv (loc (0 , 0) , cfg:sizeut) , sizeofType (} T \text{))}}$

EXTRACTFIELD-BITFIELD-START RULE: $\langle \overset{\cdot}{\text{extractField-pre (} L \text{ , bitfieldType (} T \text{ , } N \text{) , tv (bitloc (--- , } \mathit{Offset}_1 \text{ , } \mathit{Offset}_2 \text{) , ---) , tv (} \mathit{Len} \text{ , ---))}} \cdots \rangle_{\text{k}}$
$\overset{\cdot}{\text{extractField-aux (explodeToBits (} L \text{) , bitfieldType (} T \text{ , } N \text{) , } \mathit{Offset}_2 +_{Nat} \mathit{Offset}_1 *_{Nat} 8 \text{ , | truncRat (} \mathit{Len} \text{ *Rat 8) |}_{Int} \text{ , } \cdot \mathit{List\{K\}} \text{)}}$

EXTRACTFIELD-NORMAL-START RULE: $\langle \overset{\cdot}{\text{extractField-pre (} L \text{ , } T \text{ , tv (loc (--- , } \mathit{Offset} \text{) , ---) , tv (} \mathit{Len} \text{ , ---))}} \cdots \rangle_{\text{k}}$ when
$\overset{\cdot}{\text{extractField-aux (explodeToBits (} L \text{) , } T \text{ , } \mathit{Offset} *_{Nat} 8 \text{ , | truncRat (} \mathit{Len} \text{ *Rat 8) |}_{Int} \text{ , } \cdot \mathit{List\{K\}} \text{)}}$
$\mathit{getKLabel}(T) \neq_{Bool}$ bitfieldType

RULE: $\langle \overset{\cdot}{\text{extractField-aux (piece (--- , } N \text{) , } L \text{ , } T \text{ , } \mathit{Offset} \text{ , } \mathit{Len} \text{ , } \cdot \mathit{List\{K\}} \text{)}} \cdots \rangle_{\text{k}}$ when $N \leq_{Nat} \mathit{Offset}$
$\overset{\cdot}{\text{extractField-aux (} L \text{ , } T \text{ , | } \mathit{Offset} -_{Int} N \text{ |}_{Int} \text{ , } \mathit{Len} \text{ , } \cdot \mathit{List\{K\}} \text{)}}$

EXPLODE-SKIP-FLOATS RULE: $\langle \overset{\cdot}{\text{extractField-aux (explodeToBits (---) , } L \text{ , } T \text{ , sNat } \mathit{Offset} \text{ ^8 , } \mathit{Len} \text{ , } \cdot \mathit{List\{K\}} \text{)}} \cdots \rangle_{\text{k}}$
$\overset{\cdot}{\text{extractField-aux (} L \text{ , } T \text{ , } \mathit{Offset} \text{ , } \mathit{Len} \text{ , } \cdot \mathit{List\{K\}} \text{)}}$

EXPLODE-SKIP-LOCS RULE: $\langle \underline{\text{extractField-aux (explodeToBits (loc (— , —)) , } L \text{ , } T \text{ , sNat } \mathit{Offset} \text{ ^8 , } \mathit{Len} \text{ , } \cdot\mathit{List}\{K\} \text{)}} \cdots \rangle_{\text{k}}$
$\text{extractField-aux (} L \text{ , } T \text{ , } \mathit{Offset} \text{ , } \mathit{Len} \text{ , } \cdot\mathit{List}\{K\} \text{)}$

RULE: $\langle \underline{\text{extractField-aux (piece (} N \text{ , } \mathit{Size} \text{) , } L \text{ , } T \text{ , 0 , } \mathit{Len} \text{ , } L' \text{)}} \cdots \rangle_{\text{k}}$ when $\mathit{Size} \leq_{Nat} \mathit{Len}$
$\text{extractField-aux (} L \text{ , } T \text{ , 0 , | } \mathit{Len} -_{Int} \mathit{Size} \text{ |}_{Int} \text{ , } L' \text{ , piece (} N \text{ , } \mathit{Size} \text{))}$

EXPLODE-READ-FLOAT RULE: $\langle \underline{\text{extractField-aux (explodeToBits (} F \text{) , } L \text{ , } T \text{ , 0 , sNat } \mathit{Len} \text{ ^8 , } L' \text{)}} \cdots \rangle_{\text{k}}$
$\text{extractField-aux (} L \text{ , } T \text{ , 0 , } \mathit{Len} \text{ , } L' \text{ , } F \text{)}$

EXPLODE-READ-LOC RULE: $\langle \underline{\text{extractField-aux (explodeToBits (loc (} N \text{ , } M \text{)) , } L \text{ , } T \text{ , 0 , sNat } \mathit{Len} \text{ ^8 , } L' \text{)}} \cdots \rangle_{\text{k}}$
$\text{extractField-aux (} L \text{ , } T \text{ , 0 , } \mathit{Len} \text{ , } L' \text{ , loc (} N \text{ , } M \text{))}$

RULE: $\langle \underline{\text{extractField-aux (explodeToBits (} F \text{) , } L \text{ , float , 0 , — , } \cdot\mathit{List}\{K\} \text{)}} \cdots \rangle_{\text{k}}$
$\text{tv (} F \text{ , float)}$

RULE: $\langle \underline{\text{extractField-aux (} F \text{ , } L \text{ , double , 0 , — , } \cdot\mathit{List}\{K\} \text{)}} \cdots \rangle_{\text{k}}$
$\text{tv (} F \text{ , double)}$

RULE: $\langle \underline{\text{extractField-aux (} F \text{ , } L \text{ , float , 0 , — , } \cdot\mathit{List}\{K\} \text{)}} \cdots \rangle_{\text{k}}$
$\text{tv (} F \text{ , float)}$

RULE: $\langle \underline{\text{extractField-aux (} F \text{ , } L \text{ , long-double , 0 , — , } \cdot\mathit{List}\{K\} \text{)}} \cdots \rangle_{\text{k}}$
$\text{tv (} F \text{ , long-double)}$

RULE: $\langle \underline{\text{extractField-aux (— , } T \text{ , 0 , 0 , } L \text{)}} \cdots \rangle_{\text{k}}$
$\text{concretize (atv (} L \text{ , } T \text{))}$

WRITETOFD-CHAR RULE: $\langle \underline{\text{writeToFD (} \mathit{FD} \text{ , } N \text{)}} \cdots \rangle_{\text{k}} \ \langle \cdots \ \mathit{FD} \mapsto \mathit{Filename} \ \cdots \rangle_{\text{openFiles}} \ \langle \cdots \ \mathit{Filename} \mapsto \underline{S \ +_{String} \ \text{charString (} N \ \%_{Nat} \ 256 \text{)}} \cdots \rangle_{\text{files}}$
.

WRITETOFD-STRING RULE: $\langle \underline{\text{writeToFD (} \mathit{FD} \text{ , } S' \text{)}} \cdots \rangle_{\text{k}} \ \langle \cdots \ \mathit{FD} \mapsto \mathit{Filename} \ \cdots \rangle_{\text{openFiles}} \ \langle \cdots \ \mathit{Filename} \mapsto \underline{S \ +_{String} \ S'} \cdots \rangle_{\text{files}}$
.

MACRO: $0 \ |_{Nat} \ N = N$
MACRO: $\text{hex (} S \text{) = String2Rat (} S \text{ , 16)}$
MACRO: $\text{Parameter-Type-List() = } \texttt{Parameter-Type-List} \text{ (} \cdot \text{)}$
EQUATION: $\text{figureOffset (loc (} \mathit{Block} \text{ , } \mathit{Offset} \text{) , tv (} R \text{ , —) , } T \text{) = tv (bitloc (} \mathit{Block} \text{ , | } \mathit{Offset} +_{Int} \text{ truncRat (} R \text{) |}_{Int} \text{ , | truncRat (8 *Rat } R \text{ -Rat truncRat (} R \text{)) |}_{Int} \text{) , pointerType (void))}$
when T isa bitfieldType
EQUATION: $\text{figureOffset (loc (} \mathit{Block} \text{ , } \mathit{Offset} \text{) , tv (} R \text{ , —) , } T \text{) = tv (loc (} \mathit{Block} \text{ , | truncRat (7 /Rat 8 +}_{Rat} \mathit{Offset} +_{Rat} R \text{) |}_{Int} \text{) , pointerType (void))}$
when $\neg_{Bool} \ T$ isa bitfieldType
MACRO: $\text{explodeToBits (} K \text{ , } L \text{) = explodeToBits (} K \text{) , explodeToBits (} L \text{)}$
MACRO: $\text{explodeToBits (piece (} N \text{ , s}_{Nat} \ \mathit{Len} \text{)) = piece (} N \ \&_{Nat} \ 1 \text{ , 1) , explodeToBits (piece (} N \ \gg_{Nat} \ 1 \text{ , } \mathit{Len} \text{))}$
MACRO: $\text{explodeToBits (piece (} N \text{ , 0)) = } \cdot\mathit{List}\{K\}$
MACRO: $\text{explodeToBits (} \cdot\mathit{List}\{K\} \text{) = } \cdot\mathit{List}\{K\}$
END MODULE

MODULE DYNAMIC-C-SEMANTICS
IMPORTS COMMON-C-SEMANTICS
Id ::= argArray
K ::= args(*List{K}*)
| args-aux(*List{K}* , *Nat*)

K RULES:
TERMINATE RULE: $\frac{\langle \cdots \ \langle \cdots \ \langle V \rangle_{\mathsf{k}} \ \langle \cdot \rangle_{\mathsf{buffer}} \ \cdots \rangle_{\mathsf{control}} \ \cdots \rangle_{\mathsf{T}} \ \langle \cdots \ \underline{\texttt{"stdin"} \mapsto S_1 \ \texttt{"stdout"} \mapsto S_2} \ \cdots \rangle_{\mathsf{files}} \ \underline{\cdot}}{\langle V \rangle_{\mathsf{resultValue}} \ \ \underline{\cdot} \ \ \langle S_1 \rangle_{\mathsf{input}} \ \langle S_2 \rangle_{\mathsf{output}}}$

MACRO: args (*L*) = args-aux (*L* , 0)
MACRO: args-aux (*E* , *L* , *N*) = argArray [*N*] = `Cast` (`Pointer` (char) , *E*) ; $\curvearrowright$ args-aux (*L* , s_{Nat} *N*)
MACRO: args-aux (*·List{K}* , *N*) = argArray [*N*] = `Cast` (`Pointer` (char) , NULL) ;
MACRO: eval (*P*) = eval (*P* , *·List{K}* , "")
MACRO: eval (*P* , *L* , *Input*) = $\langle \cdots \ \langle \cdots \ \langle P \curvearrowright$ `Global` (Declaration (char , `Pointer` (argArray [s_{Nat} $length_{ListK}(L)$]))) $\curvearrowright$ args (*L*) $\curvearrowright$ `Apply` (main , $length_{ListK}(L)$, argArray)$\rangle_{\mathsf{k}}$ $\cdots\rangle_{\mathsf{control}}$ $\cdots\rangle_{\mathsf{T}}$ $\langle$"`stdin`" $\mapsto$ *Input* "`stdout`" $\mapsto$ ""$\rangle_{\mathsf{files}}$
END MODULE