

A FORMAL SEMANTICS OF C WITH APPLICATIONS

BY

CHARLES MCEWEN ELLISON III

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Associate Professor Grigore Roşu, Chair and Director of Research
Professor Gul Agha
Professor José Meseguer
Principal Researcher Wolfram Schulte, Microsoft Research

Abstract

This dissertation shows that complex, real programming languages can be completely formalized in the \mathbb{K} Framework, yielding interpreters and analysis tools for testing and bug detection. This is demonstrated by providing, in \mathbb{K} , the first complete formal semantics of the C programming language. With varying degrees of effort, tools such as interpreters, debuggers, and model-checkers, together with tools that check for memory safety, races, deadlocks, and undefined behavior are then generated from the semantics.

Being executable, the semantics has been thoroughly tested against the GCC torture test suite and successfully passes 99.2% of 776 test programs. The semantics is also evaluated against popular analysis tools, using a new test suite in addition to a third-party test suite. The semantics-based tool performs at least as well or better than the other tools tested.

Acknowledgments

I would like to thank my family for supporting me and always being there when I needed help. In particular, my father for giving me math problems to solve and introducing me to interesting articles in the encyclopedia; my mother for believing in me and always pushing me to do my best; my aunt and uncle, Janet and Danny, for giving me my first taste of computers when they let me play with their Kaypro II; my uncle Tim for letting me have all of his old computer books; and everyone else for being so loving and caring.

I would also like to thank my adviser Grigore Roşu. Grigore's creation of \mathbb{K} , and his continued dedication and passion for the framework, made my work possible. He always sought the best solution to every problem and he never made me feel dumb for asking dumb questions. I would also like to thank the rest of my committee—Gul Agha, José Meseguer, and Wolfram Schulte—whose thoughtful contributions helped strengthen my thesis.

I never would have made it this far without my colleagues. Some of the many who have touched my life while a grad student include: Nick Chen, Dennis Griffith, Mark Hills, Michael Ilseman, Dongyun Jin, Mike Katelman, David Lazar, Choonghwan Lee, Pat Meredith, Maurice Rabb, Ralf Sasse, and Andrei Ştefănescu. In particular, I would like to thank Paul Adamczyk and Traian Şerbănuţă, whose kind words and patient support were my rock in so many difficult times. Thank you all for making each day a little brighter.

My friends outside of school—Amie, Dixie, Heidi, Jeff, Megan, Miroslav, Patrick, Rebecca, Robyn, Toria, Winna—thank you for listening to me complain without complaint. Thanks for giving sound advice. Thanks for being there when I felt down. Thank you.

My research has been supported in part by NSA contract H98230-10-C-0294, by NSF grants CCF-0916893, CNS-0720512, and CCF-0448501, by NASA contract NNL08AA23C, by a Samsung SAIT grant, by several Microsoft gifts, and by (Romanian) SMIS-CSNR 602-12516 contract no. 161/15.06.2010.

Table of Contents

Chapter 1	Introduction	1
1.1	Problem Description and Contribution	1
1.2	Why Details Matter	5
Chapter 2	Related Work	8
2.1	Comparison with Existing Formal C Semantics	8
2.2	Other Formal Semantics	13
2.3	Semantics and Formal Analysis Tools	19
Chapter 3	Background	22
3.1	C Standard Information	22
3.2	Rewriting Logic and \mathbb{K}	24
Chapter 4	Positive Semantics	29
4.1	Introduction	29
4.2	The Semantics of C in \mathbb{K}	30
4.2.1	Syntax	30
4.2.2	Configuration (Program + State)	31
4.2.3	Memory Layout	32
4.2.4	Basic Semantics	33
4.2.5	Static Semantics	41
4.2.6	Concurrency Semantics	43
4.2.7	Parametric Behavior	47
4.2.8	Expression Evaluation Strategy	48
4.2.9	Putting It All Together with <code>kcc</code>	49
4.3	Testing the Semantics	51
4.3.1	GCC Torture Tests	51
4.3.2	Exploratory Testing	53
4.4	Applications	53
4.4.1	Debugging	54
4.4.2	State Space Search	55

Chapter 5	Negative Semantics	61
5.1	Introduction	61
5.2	Undefinedness	63
5.2.1	What Undefinedness Is	63
5.2.2	Undefinedness is Useful	64
5.2.3	Undefinedness is also a Problem	65
5.2.4	Strangeness of C Undefinedness	66
5.2.5	Implementation-Dependent Undefined Behavior	67
5.2.6	Difficulties in Detecting Undefined Behavior	69
5.2.7	Undefinedness in Other Languages	70
5.3	Semantics-Based Undefinedness Checking	71
5.3.1	Using Side Conditions and Checks to Limit Rules	72
5.3.2	Storing Additional Information	76
5.3.3	Symbolic Behavior	80
5.3.4	Suggested Semantic Styles for Undefinedness	83
5.4	Applications	85
5.4.1	A Semantics-Based Undefinedness Checker	85
5.4.2	State Space Search Revisited	89
5.5	Evaluation	93
5.5.1	Third Party Evaluation	93
5.5.2	Undefinedness Test Suite	97
5.6	Conclusion	99
Chapter 6	Conclusion	101
6.1	Limitations	101
6.2	Future Work	103
6.2.1	Semantics	103
6.2.2	Tools	103
6.3	Conclusion	104
Appendix A	Entire Annotated Semantics	106
A.1	Syntax	107
A.2	Configuration	127
A.3	Expressions	130
A.4	Statements	178
A.5	Typing	202
A.6	Declarations	244
A.7	Memory	285
A.8	Standard Library	309
A.9	Error Handling	345
A.10	Miscellaneous	364
References		409

Chapter 1

Introduction

This dissertation shows that complex, real programming languages can be completely formalized in the \mathbb{K} Framework, yielding interpreters and analysis tools for testing and bug detection. This is demonstrated by providing, in \mathbb{K} , the first complete formal semantics for C. From this definition, we extract a number of useful tools.

In this chapter, we explain the problem context and our particular contributions. In Chapter 2 we provide a detailed comparison with other formal semantics of C, a brief overview of formal semantics for other programming languages, and a brief comparison of different formalisms. Chapter 3 gives background material necessary for understanding the rest of the dissertation, including information on C and on the \mathbb{K} framework. Chapter 4 focuses on the semantics of defined programs, including how the semantics is organized, how we evaluated it, and what tools are generated. Chapter 5 does the same for undefined programs—it describes the consequences of having undefinedness in C as well as what formal techniques are needed to identify such programs. An evaluation and investigation of tools is also done in Chapter 5. Chapter 6 summarizes our work (including limitations) and suggests possible avenues of research building on it. Finally, Appendix A contains the entire dynamic semantics, annotated with excerpts from the C standard.

1.1 Problem Description and Contribution

Programming language semantics and program analysis are well developed research areas with a long history. Many definitional formalisms have been suggested over the years, each capturing the imagination of groups of researchers with varying degrees of success. However, despite the abundance of formalisms, programming languages are not designed or analyzed using formal semantics as a matter of course. Very few languages have been formally

defined in their entirety, and very few analysis tools are based on even the semantic subsets that exist.

This lack of semantics-based approach to software analysis is curious considering that it has been a goal of much of the research in the area in the last 30 years [19, 26, 60, 72, 107, 127]. The benefits of such a system would be numerous, and include:

- when multiple tools for a language are based off of a single semantics of that language, one gains confidence that the tools themselves are trustworthy;
- it is simply more practical to describe the semantics in a single place instead of many times across multiple tools;
- tools simply cannot be shown to be sound without a formal semantics against which to compare them.

Given that the semantics-based approach to programming languages and analysis tools has such strong advantages, there must be some reason why it has not become mainstream. One theory is that each of the prevailing formalisms suffer serious weaknesses, making them less than suitable for the task [146]. One recent formalism whose goal is to meet these challenges and that has been gaining traction is the \mathbb{K} Framework [134, 148], a rewriting-based formalism for defining programming languages, type systems, and calculi. \mathbb{K} has shown much promise, having been used to define the semantics of many academic languages [73, 146, 147, 150], as well as subsets of a handful of real programming languages [27, 53, 100, 102, 138]. \mathbb{K} has many good properties, including the ability to specify language rules modularly and represent truly concurrent programming languages faithfully [146]. While such properties are necessary for a definitional framework capable of fulfilling this dream, they are not sufficient. Until now, \mathbb{K} had not been used to describe the complete semantics of any real programming language. It is possible that, while being quite effective at describing toy languages or language subsets, the complexity of a real language could overwhelm the capabilities of \mathbb{K} .

We have allayed these fears with our definition of the C programming language. C is a popular, complex language that 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 [80]. 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.

Despite its continuing popularity for over 40 years, no complete formal semantics for C was previously given in any formalism (Section 2.1). Our definition of C in \mathbb{K} thus represents the first complete formal semantics of C—we say complete, in the sense that it covers the semantics of all correct programs as defined by the C standard (see Sections 3.1 and 4.1 for more details). We tested our semantics against the regression tests used by the GCC compiler and found that our semantics ran 99.2% of the tests correctly, which is better than GCC itself or Clang, and only one test fewer than ICC (Section 4.3). Such a definition in \mathbb{K} , produced in about a year and a half of individual effort, proves that \mathbb{K} is capable of representing the semantics of real programming languages in all their detail. We discuss these semantics in detail in Chapter 4.

In addition to the semantics of correct programs (which we call the *positive* semantics), we have also formalized a significant portion of the *negative* semantics of C, that is, the ability to identify semantically invalid programs. Such a semantics allows one to determine whether or not a C program contains undefined behavior, such as a division by zero. We show that this problem is undecidable in theory, but it can often be answered in practice. The negative semantics is discussed in depth in Chapter 5.

Faithful definitions alone do not solve the problem of semantics-based analysis tools. There remains the issue of the tools themselves. We have created a suite of tools directly from the single C semantics; of course we build on the work of many others, including works specifically related to \mathbb{K} [7, 50, 53, 74, 133, 134, 138, 139, 146, 148] as well as the underlying rewriting-logic theories to which \mathbb{K} is compiled [29, 30, 46, 96, 104, 132]. These tools include interpreters, debuggers, state-space search tools, and

model-checkers, together with tools that check for memory safety, races, and deadlocks. The positive semantics is enough for tools that explore the behaviors of correct programs, while the negative semantics is necessary for identifying or preventing incorrect behavior. Not only are these tools possible, but we have shown that the tools are usable—a tool to identify and report undefined behavior has been used as a component in work on test case reduction (Section 5.4.1).

Contributions The specific contributions of this dissertation include:

- a detailed comparison of other C formalizations;
- the most comprehensive formal semantics of C to date, which is executable and has been thoroughly tested;
- demonstrations as to its utility in exploring program behavior;
- constructive evidence that rewriting-based semantics scale;
- a systematic formal analysis of undefinedness in C;
- identification and comparison of techniques that can be used to define undefinedness;
- a semantics-based tool for identifying undefined C programs;
- initial work on a test suite for undefined behavior in C.

The tool, the semantics, and the test suite can all be found at <http://c-semantics.googlecode.com/>.

Features Our semantics captures every feature required by the C99 standard. We include a partial list here to give an idea of the completeness, and explain any shortcomings in Section 6.1. All aspects related to the below features are included and are given a direct semantics (e.g., not translated to other features using a parser or other informal frontend):

- Expressions: referencing and dereferencing, casts, array indexing (`a[i]`), structure members (`->` and `.`), arithmetic, bitwise, and logical operators, `sizeof`, increment and decrement, assignments, sequencing (`_,_;`), ternary conditional (`_?_:_;`);

- Statements: `for`, `do-while`, `while`, `if`, `if/else`, `switch`, `goto`, `break`, `continue`, `return`;
- Types and Declarations: `enums`, `structs`, `unions`, bitfields, initializers, static storage, `typedefs`, variable length arrays;
- Values: regular scalar values (signed/unsigned arithmetic and pointer types), `structs`, `unions`, compound literals;
- Standard Library: `malloc/free`, `set/longjmp`, basic I/O;
- Conversions: (implicit) argument and parameter promotions and arithmetic conversion, and (explicit) casts.

1.2 Why Details Matter

Many features of C are crosscutting, in that their semantics have potential ramifications for other features in the language. For example, supporting bitfields might require changing the way memory is handled—suddenly, instead of every access to memory being at the byte level, some are sub-byte or across byte boundaries. Similarly, the memory returned by `malloc()` is a different “kind” of memory than that created for local or global variables [81, §6.2.4, §6.5:6, & §7.22.3]. Not only is the lifetime different, but the types of objects stored in the memory can change. One final example is `setjmp()` and `longjmp()`, which allow a programmer to save the program context and later return to it. What counts as context changes depending on what features of C are supported by an implementation. In many definitional frameworks, it is likely that features would need to be written with consideration given to their effect on `longjmp()`.

Other features, while not crosscutting, are surprisingly intricate. It is tempting to gloss over the details of C’s arithmetic and other low-level features when giving it a formal semantics. However, C is designed to be translatable to machine languages where arithmetic is handled by any number of machine instructions. The effects of this overloading 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. Here we give a few examples that reveal even apparently simple C programs can involve complex semantics.

For the purposes of these examples, assume that `ints` are 2 bytes (capable of representing the values -32768 to 32767) and `long ints` are 4 bytes (-2147483648 to 2147483647). Also, unless specified, in C a type is assumed to be signed.¹ In the following program, what value does `c` receive [158, 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 `ints`, 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.

What if we make 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. The result is computed by simply reducing the value modulo one more than the max value [81, §6.3.1.3:2]. $1000000 \bmod 65536$ gives us 16960.

One last variation—`signed chars` are one byte in C (-128 to 127).² 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 `ints` are promoted to `ints` before doing arithmetic. There are essentially 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

¹Except `chars` and bitfields, whose signedness is implementation-defined.

²Bytes are only required to be at least 8 bits long.

wants to find bugs in real programs. Even though errors related to the above details continue to be found in real compilers [168], previous semantics for C either did not give semantics at this level of detail, or were not suitable for identifying programs that misused these features. This is one of our primary reasons for wanting an executable semantics.

As seen in the next chapter, more than half of existing C semantics leave out details related to arithmetic, conversions, or bitfields, and none handle `malloc()` or `longjmp()`.

Chapter 2

Related Work

There is an enormous amount of work in the area of formal semantics and program analysis. Here we look at existing formal semantics of programming languages, in addition to definitional frameworks that come with tool support. We start with a particular focus on existing semantics of C.

2.1 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 (1993) One of the earliest formal descriptions of ANSI C is given by Gurevich and Huggins [67], 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 [88]. 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 [170], 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 1.2), 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 [119, p. 11], their semantics was constructed without the benefit of any mechanization. According to Gurevich,¹ 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 (1994) Soon after the previous definition, Cook et al. [33] describe a denotational semantics of C90 using a custom-made temporal logic for the express purpose of proving properties about C programs. 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 (1994) The work of Cook and Subramanian [32, 157] is a semantics for a restricted subset of C, based loosely on the semantics above. This semantics is embedded in the theorem prover Nqthm [21] (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 *n*th element of the array `a` into `p` [32, p. 122]. Their semantics is, at its roots, an interpreter—it uses a similar technique to that

¹Personal communication, 2010.

²Personal communication, 2010.

described by Blazy and Leroy [13] 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 (1998) The next major semantics was provided by Norrish [119], 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++ [120]). His semantics is in the Structural Operational Semantics (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. In Sections 4.2.8 and 4.4.2 we demonstrate how our definition captures the same behaviors.

Working inside HOL provides an elegant solution to the underspecification of the standard—Norrish can state facts given by the standard as axioms/theorems. For example, the standard says that the number of bits in a byte is defined by the macro `CHAR_BIT`, and must be *at least* 8. In turn, Norrish’s semantics contains the theorem $\vdash \text{CHAR_BIT} \geq 8$ [119, p. 30], which describes precisely this information. In general, he pays a lot of attention to underspecification, using HOL to his advantage, as above. To maintain executability, we chose instead to parameterize our definition for those implementation-defined choices. 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 programs (see Section 4.4.2). Proofs of program correctness [139] as well as semantics-level proofs [50] 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 (≤ 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 (2001) A denotational semantics for C99 is described by Papaspyrou [124, 125] 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 six or the fourth Fibonacci number.

Blazy and Leroy (2009) A big-step operational semantics for a subset of C, called Clight, is given by Blazy and Leroy [13]. 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. Their semantics includes coinductive rules for divergence, enabling proofs of non-termination or properties of non-terminating programs, which traditionally has been difficult with big-step semantics.

To help validate their 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 was eventually completed in version 1.9 of the CompCert C compiler [77], though we have not had time to evaluate it against our tools.

Clight does not handle non-determinism or sub-expressions with side effects. However, since publication, they have added a new front-end small-step definition called CompCert C that does handle these features, and is also being used to handle `goto`.³

³Personal communication, 2011.

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

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

GH represents Gurevich and Huggins [67], *CCR* is Cook et al. [33], *CR* is Cook and Subramanian [32], *No* is Norrish [119], *Pa* is Papaspyrou [125], *BL* is Blazy and Leroy [13], and *ER* is our work.

Figure 2.1: Dynamic Semantics Features

There are other formal semantics of C (or fragments of C) that we choose not to review here, including Black [12], Bofinger [15], and Bortin et al. [20], 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 in Figure 2.1. 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, although not all features are equally difficult.

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 [81, §6.5.2.1:3], 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.

2.2 Other Formal Semantics

Other languages than C, of course, have also been formalized to varying degrees of completeness. We take a brief look at many of these other semantics in this section in order to gain a perspective of where the C definitions lie. Due to the already vast numbers of existing semantics, we limit our focus to general purpose programming languages and to semantics written in the last 30 years.

Looking through these semantics paints a broad picture of the state of the art in language formalization. First, nearly all of the semantics are incomplete in some way. Those that are complete are either complete by definition (e.g., the official definition of SML [109]) or are for relatively small languages (e.g., the definitions of Prolog). Further, very few have been used for any purpose other than to push the various formalisms or techniques. A few have been used to assist the development of the languages they define, or at least provide formal evidence that certain features need changes [37, 166]. Others have been used for proofs about either the language or programs in the language [44, 95, 120, 142, 161, 169]. Finally, less than one third yielded tools either directly or with a little extra work. Three of these were definitions that use the \mathbb{K} framework, the same formalism we use for our semantics of C.

This chart makes it clear that programming languages, at least in part, can be formalized. It is much less clear that formal semantics can be useful; we hope that our focus on tool support (Sections 4.4 and 5.4) may address this weakness.

Language	Year	Work	Formalism	Machn.	Exec.	Description
Java	1990	Attali, Caromel, and Russo [8]	Centaur & TYPOL (big-step and small-step SOS)	✓	✓	Generates programming environment; excludes exceptions, arrays, packages
	1997	Comstedt and Holmén [31, 75]	RML	✓	✓	Started as Java 1.0, extended to cover most of 1.2; contains formal, executable translation to JVM
	1997	Wallace [163]	ASMs with Montages	×	×	Covers Java 1.0; excludes volatiles and definite assignment of variables before use
	1999	Alves-Foss and Lam [2]	Denotational	×	×	specification based (1996); excludes concurrency and APIs
	1999	Börger and Schulte [17]	ASMs	×	×	Standards-based; leaves out packages, I/O, <code>for</code> , <code>do</code> , <code>switch</code> , loading/linking classes, and GC
	1999	Brown and Watt [22]	Action Semantics	×	×	Covers much of Java 1.0; leaves out concurrency
	2000	Drossopoulou, Valkevych, and Eisenbach [43, 44]	Small-step, RSwEC for Exceptions	×	×	Used for proof of type soundness; leaves out library, arithmetic
	2001	Stärk, Schmid, and Börger [155]	ASMs in AsmGofer	✓	✓	Comes with GUI for exploring evaluation
2004	Farzan, Chen, Meseguer, and Roşu [27, 53]	\mathbb{K} -style	✓	✓	583 rules; used for model checking concurrent programs; leaves out certain features like <code>super</code> , bit-wise operations, and the library	

Language	Year	Work	Formalism	Machn.	Exec.	Description
C++	1993	Wallace [162]	ASMs	×	×	Written before standard existed
	2008	Norrish [120]	Small-step in HOL	✓	×	Used to prove simple “sanity” theorems about language and theorems about behavior of concrete programs; ignores overloading, function pointers, <code>enums</code> , <code>typedefs</code> , <code>unions</code> , bit fields, <code>goto</code> , and <code>switch</code>
C#	2003	Börger, Fruja, Gervasi, and Stärk [18], Jula and Fruja [85]	ASMs	✓	✓	Standards-based
SML	1990	Milner, Tofte, Harper, and MacQueen [108, 109]	Big-step	×	×	186 rules; official standard for SML; complete (by definition)
	1994	VanInwegen and Gunter [161]	Big-step in HOL	✓	×	Used semantics to prove some properties about language (e.g., determinism); focuses on dynamic semantics of the Core and ignores reals
	1994	Maharaj and Gunter [95]	Big-step in HOL	✓	×	Adds support for Module system to above work; used to prove that this is a conservative extension; focuses on dynamic evaluation only

Language	Year	Work	Formalism	Machn.	Exec.	Description
	1999	Watt [164]	Action Semantics	×	×	Author argues his semantics is more readable, modular, and formal than official standard; does not give static semantics for the Module layer
	2000	Cater and Huggins [24]	ASMs	×	×	Defines only dynamic semantics of the core
LLVM	2012	Zhao, Nagarakatte, Martin, and Zdancewic [169]	Small-step SOS	✓	✓	Formalized in Coq; used to prove program transformation correct; leaves out exceptions, variadic argument functions, vector types
SmallTalk	1987	Wolczko [166]	Denotational	×	×	Used to argue about language design; does not handle concurrency, global variables, difficult details of blocks, or machine ints
	1992	Deransart and Ferrand [37]	Logic Programming	✓	✓	Used in the standardization of Prolog
Prolog	1995	Börger and Rosenzweig [16]	ASMs	×	×	Standards-based; used for reasoning about implementations and clarifying disputable features; nearly complete—leaves out only syntax, OS interface, and arithmetic
	2000	Kulaš and Beierle [90]	Rewriting Logic	✓	✓	Continuation based; complete

Language	Year	Work	Formalism	Machn.	Exec.	Description
	2011	Ströder, Emmes, Schneider-Kamp, Giesl, and Fuhs [156]	Linear inference rules	×	×	153 rules; standards-based; used for termination analysis; complete
	2007	Meredith, Hills, and Roşu [100, 101]	\mathbb{K} -style	✓	✓	682 rules; incomplete support for macros
Scheme	2010	Sperber, Dybvig, Flatt, van Straaten, Findler, and Matthews [154]	RSwEC in PLT Redex	✓	✓	Does not support macros, I/O, or the numerical tower; handles some underspecified behaviors
	1998	Pace and He [123]	Relational Duration Calculus	×	×	Used to prove properties about different algorithms; leaves out major features like non-blocking assignments
Verilog	1999	Sasaki [142]	ASMs	×	×	Deterministic
	2010	Meredith, Katelman, Meseguer, and Roşu [102]	\mathbb{K} -style	✓	✓	582 rules; used to run programs and search behaviors; does not support analog features, tasks, or functions

Language	Year	Work	Formalism	Machn.	Exec.	Description
XQuery & XPath	2010	Draper, Fankhauser, Fernández, Malhotra, Rose, Rys, Siméon, and Wadler [42]	Big-step	×	×	442 rules; gives both static and dynamic semantics; complete (by definition)
Python	2009	Smeding [153]	Small-step in Haskell	✓	✓	Used to run small programs; leaves out standard library, garbage collection, threading, FFI, and reflection
	1992	Jones and Wadler [84]	Big-step	×	×	Static semantics only; gives translation into a language without overloading; leaves out a few features such as constant and n+k patterns
Haskell	1993	Hammond and Hall [69]	Big-step	×	×	83 rules; builds on above static semantics; covers most of Haskell 1.0; leaves out definitions of most <i>PreludeCore</i> functions
	2002	Faxén [54]	Big-step	×	×	Static semantics only; does not handle ambiguous overloading and its resolution, newtype declarations, or deriving clauses in ADT declarations

2.3 Semantics and Formal Analysis Tools

As our goal is to explore how a complete language can be defined in \mathbb{K} , we do not focus much on other formalisms. However, here we give a brief (and incomplete) list of other frameworks from the perspective of supported tools. For a more in-depth analysis of competing formalisms and how they relate to \mathbb{K} and rewriting-based formalisms, please see Şerbănuță [146] and Şerbănuță et al. [149].

Probably the work that comes closest, from a practical standpoint, to the kind of tool suite we are aiming for would be Frama-C [25, 34]. Frama-C is a suite of analysis tools for C based around a plug-in architecture, including tools for value analysis, deductive verification, LTL specification verification, slicing, and more. The tools are not semantics-based, although each individual analysis can query the other tools, which allows for less redundancy among the tools together. However, because the tools are based on informal models of the language, they can often allow you to prove false things. Using Caduceus [56, 57], the predecessor of Frama-C,⁴ we were able to “prove” correct the following program:

```
int abs(int x){
    if (x >= 0) { return x; } else { return -x; }
}
```

At first glance, this program does look correct. However, on most systems utilizing two’s complement arithmetic, $-\text{INT_MIN} = \text{INT_MIN}$. This is obvious if you think about it, as (for a two byte `int`) $-(-32768)$ cannot be 32768, because 32768 is not even in the set of possible signed two byte numbers. In fact, this is a signed overflow, and by the semantics of C, is actually undefined. Simply evaluating $-\text{INT_MIN}$ results in a signed overflow, which is undefined in C. This could result in any behavior, including looping or crashing the computer. Because the model of integers used in the Caduceus tool was too simple, it allowed an incorrect program to be verified.

There has been a lot of work done specifically in generating interpreters and compilers from formal definitions. Andrews et al. [3] describes an interpreter derived from a formal definition of Modula-2, but we could not find any evidence that they completed work on the proposed algorithms to automatically

⁴We have not attempted to re-verify this result in the newest version of Frama-C; it is only being used here as an example of a larger methodological issue.

generate the interpreter. CENTAUR [19] is an older system that can generate interpreters from formal specifications of a language. They experimented with using both ASF [11] and “Natural Semantics” [86] (big-step SOS). Although big-step definitions lend themselves to executability, they lack many other features useful in a definitional framework such as modularity or concurrency, which \mathbb{K} handles naturally.

The ASF+SDF Meta-Environment [38], a successor to CENTAUR, supports the ASF+SDF Compiler [160]. This compiler can translate specifications written in the ASF+SDF framework to C code which can then be compiled and run natively. Their framework does not support full matching modulo commutativity, which means it is difficult to ignore order of program state. This, in turn, may lead to less modular definitions. Additionally, ASF+SDF has no concept of rules as in rewriting logic, so there is no natural way to represent that certain operations are concurrent.

The LISA system [103] can generate compilers and interpreters (as well as a number of other useful tools) from FSM and attribute grammar descriptions of programming languages. However, their formal specification language is fairly limited—while the attribute grammars can be used to specify some simple semantic constructs, any moderately difficult construct (assignment, conditionals, etc.) is specified informally in Java. With this in mind, it is understandable that they are able to execute specifications but also raises questions about the formality of much of their semantics.

The Maude system [30], to which \mathbb{K} is currently compiled for executability and search, can also be used directly as a platform for semantic development [105, 106, 149]. Maude allows users to describe rewriting logic [104] theories, yielding interpreters, state-space search tools, model checkers [46], and debuggers [132]. Maude has been used to give a complete semantics of Prolog [90]. It has also been used to define the semantics of much of the C preprocessor [64].

Like \mathbb{K} , Maude serves as the basis for an MSOS [112] platform called Maude-MSOS [26, 36]. With the Maude backing, the Maude-MSOS tool yields executable specifications and the same kinds of analysis tools available under Maude directly. MSOS was designed to address the inherent modularity issues in traditional SOS specifications, where feature additions or changes would require changes to unrelated features.

The relational meta-language (RML) can also be used in the generation of

interpreters and development environments from formal specifications [60]. Their tool takes operational (mostly big-step) specifications of programming languages in RML and translates them to efficient C code, complete with optimizations. For the Mini-Freja language, the generated code was significantly faster than that of hand-written Prolog code and at least 10,000 times faster than the interpreter obtained from the Maude-MSOS tool [26]. There do not appear to be benchmarks against any languages with direct implementations.

The ASM formalism [66] has been wildly successful, with many languages defined, including Java [155] and C# [18] (many others listed in the table found in the previous section). Their success has spawned a number of semantic tools, including ASMETA [63], AsmL [68], CoreASM [52], AsmGofer [143], TASM [121], and XASM [4]. These systems take ASM specifications and yield interpreters, GUIs for inspecting execution traces, model checkers, and test-case generators. The ASM approach continues to be developed, with recent work on service-oriented architectures [131] and runtime monitoring [6].

Another system for mechanized formal semantics is the PLT Redex system [55, 97]. Redex has been used to define a large subset of Scheme [154] as well as Datalog [98]. Their system yields tools for debugging definitions, testing, execution, and exploration of state-space. Many of the same tools are available for users of \mathbb{K} , so the differences between the two systems lie mostly in the supported formalisms used (reduction semantics with evaluation contexts for Redex, and \mathbb{K} for the \mathbb{K} system).

The Ott [152] tool is a semanticist's assistant, helping with typesetting and lightweight consistency checks of a definition. The Ruler [40] tool is similar, though targetted toward type systems. It can additionally generate an executable implementation based on attribute grammars. A thorough comparison of Redex, \mathbb{K} , Ott [152], and Ruler [40], as well as the additional tools α ML [91] and α Prolog [28] can be found in Klein et al. [89].

Chapter 3

Background

In this chapter we give a little background on the C standard, including some important definitions. We additionally explain \mathbb{K} , the rewriting formalism in which we give our semantics of C.

3.1 C Standard Information

The C programming language, developed in the early 1970s by Dennis Ritchie [88], has become one of the most important programming languages to date. It has influenced countless other languages and is still widely used. According to 2010 usage information for open source projects on Google Code [165], C is used by more projects than any other programming language. It also stands at the top of the TIOBE index of programming language popularity [159] in May 2012, and has been in either of the top two positions since 2001.

The popularity of C meant many competing compilers have been developed over the years [83, p. 14]. To ensure programs written for one compiler would work when compiled using a different compiler, a standard version of C needed to be created, which defined the language. According to the forward of the ANSI C standard [167]:

The need for a single clearly defined standard had arisen in the C community due to a rapidly expanding use of the C programming language and the variety of differing translator implementations that had been and were being developed. The existence of similar but incompatible implementations was a serious problem for program developers who wished to develop code that would compile and execute as expected in several different environments.

Part of this problem could be traced to the fact that implementors did not have an adequate definition of the C language upon which to base their implementations. The *de facto* C programming language standard, *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, is an excellent book; however, it is not precise or complete enough to specify the C language fully. In addition, the language has grown over years of use to incorporate new ideas in programming and to address some of the weaknesses of the original language.

[ANSI] C addresses the problems of both the program developer and the translator implementor by specifying the C language precisely.

The ANSI C standard was completed in 1989, and was quickly turned into the ISO C90 standard [78] in 1990. The C standard has been an evolving entity, and C90 was bolstered with two technical corrigenda (ISO/IEC 9899 TCOR1 and ISO/IEC 9899 TCOR2) and an amendment (AMD1), which together with the C90 standard make the C95 standard [80, p. 1]. The C99 standard [79] was the latest C standard between 1999 and 2011, when it was finally replaced by the C11 standard [81]. C99 added many features to C, including `restricted` pointers, variable length arrays, flexible array members, `long long int`, compound literals, and designated initializers. C99 also reworked many features, such as rules of effective type, integer division and modulus, and integer promotion rules, in order to clarify their meanings. Finally, C99 removed certain behaviors, such as implicit `int` declarators or function declarations. C11 continued adding features and clarifying the language, although with arguably more restraint than the C99 committee (15 major changes in C11 as opposed to 54 in C99 [81, pp. xiii–xvi]). Most notably, it adds support for concurrency, something that was previously handled by external library calls to things like POSIX threads [76].

The C standard uses the idea of undefined and partially defined behaviors in order to avoid placing difficult requirements on implementations. It categorizes the particular behaviors of any C implementation that are not fully defined into four categories [81, §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 [data, with] no requirements.

locale-specific behavior Behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

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. An example of locale-specific behavior is whether `islower()` (is-lower-case) returns true for characters other than the 26 lowercase Latin letters. In this dissertation, we focus on the first three such behaviors and consider the fourth as implementation-defined.

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” [81, §4.5]. This is called “strictly conforming”. However, programmers use C for many inherently non-portable tasks, such as writing device drivers. The standard offers another level of conformance (called “conforming”) where the program may rely on implementation-defined or even unspecified (but never undefined) behavior. Based on this, our definition is parametric in implementation-defined behaviors (Section 4.2.7), and uses symbolic computation to describe unspecified behaviors (Section 5.3.3).

3.2 Rewriting Logic and \mathbb{K}

To give our semantics, we use a rewriting-based semantic framework called \mathbb{K} [134], inspired by rewriting logic (RL) [104]. In particular, our semantics is written using the K-Maude tool [148, 150], which takes \mathbb{K} rewrite rules and translates them into Maude [30]. Maude is a rewriting-logic engine that provides facilities for the execution and analysis of rewriting-logic theories.

RL organizes term rewriting *modulo equations* (namely associativity, commutativity, 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 languages is that the evolution of a program can be clearly described using rewrite rules. A rewriting theory consists essentially of a signature describing terms and a set of rewrite rules that describe steps of computation. Given some term allowed by signature (e.g., a program together with input), deduction consists of the application of the rules to that term. This yields a transition system for any program. A single path of rewrites describes the behavior of an interpreter, while searching all paths would yield all possible answers in a nondeterministic program.

For the purposes of this dissertation, 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 labeled, nested multisets, as seen in Figure 3.1. 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 typical rule for a simple imperative language (see Section 4.2.4 for the equivalent rule in C) for dereferencing a variable:

$$\frac{\langle * X \dots \rangle_{\mathbf{k}} \quad \langle \dots X \mapsto L \dots \rangle_{\mathbf{env}} \quad \langle \dots L \mapsto V \dots \rangle_{\mathbf{mem}}}{V}$$

We see here three cells: `k`, `env`, and `mem`. The `k` cell represents a list (or stack) of computations waiting to be performed. The left-most (i.e., top) element of the stack is the next item to be computed. The `env` cell is simply a map of variables to their locations. The `mem` cell is a map of locations to their values. The rule above says that if the next thing to be evaluated (which here we call a redex) is the application of the dereferencing operator (`*`) to a variable `X`, then one should match `X` in the environment to find its location `L` in memory, then match `L` in memory to find the associated value `V`. With this information, one should transform the redex into `V`.

This example exhibits a number of features of \mathbb{K} . First, rules only need to mention those cells (again, see Figure 3.1) relevant to the rule. The rest of the cell infrastructure can be inferred, making the rules robust under most extensions to the language. 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`,

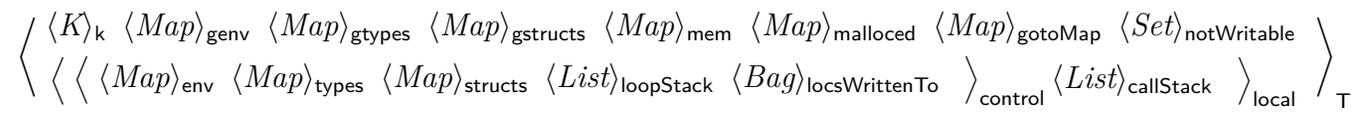


Figure 3.1: Subset of the C Configuration

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 change.

This unconventional notation is quite useful. The above rule, written out as a traditional rewrite rule, would be:

$$\begin{aligned} \langle * X \curvearrowright \kappa \rangle_k \langle \rho_1, X \mapsto L, \rho_2 \rangle_{\text{env}} \langle \sigma_1, L \mapsto V, \sigma_2 \rangle_{\text{mem}} \\ \Rightarrow \langle V \curvearrowright \kappa \rangle_k \langle \rho_1, X \mapsto L, \rho_2 \rangle_{\text{env}} \langle \sigma_1, L \mapsto V, \sigma_2 \rangle_{\text{mem}} \end{aligned}$$

Items in the k cell are separated with “ \curvearrowright ”, which can now be seen. The κ and $\rho_1, \rho_2, \sigma_1, \sigma_2$ take the place of the “ \dots ” above. The most important thing to notice is that nearly the entire rule is duplicated on the right-hand side. Duplication in a definition requires that changes be made in concert, in multiple places. If this duplication is not kept in sync, it leads to subtle semantic errors. In a complex language like C, the configuration structure is much more complicated, and would require actually including additional grouping cells like `control` and `local` (Figure 3.1) in the rule. These intervening cells are automatically inferred in \mathbb{K} , which keeps the rules more modular.

Notationally, we use “ \cdot ” to represent the unit element of any algebraic lists or sets (including the “ \curvearrowright ” list). We also use “ $-$ ” to stand for a term that we do not care to name. Finally, in order to get the redexes to the top of the k cell (i.e., in order to identify which positions in the syntax tree can be reduced next), the grammar of C is annotated with additional “strictness” annotations. For example, for addition, we say that

$$Exp ::= Exp + Exp \text{ [strict]}$$

meaning that either argument of the addition operator can be taken out for evaluation, nondeterministically. In contrast, the `if` construct looks like this:

$$Stmt ::= \text{if } (Exp) \text{ Stmt } \text{ [strict(1)]}$$

indicating that only the first argument can be taken out for evaluation. The two annotations above cause the following six rules to be automatically

generated:

$$\begin{array}{c|c|c}
 \langle \frac{E_1 + E_2}{E_1 \curvearrowright \square + E_2} \dots \rangle_k & \langle \frac{E_1 + E_2}{E_2 \curvearrowright E_1 + \square} \dots \rangle_k & \langle \frac{\text{if } (E) S}{E \curvearrowright \text{if } (\square) S} \dots \rangle_k \\
 \langle \frac{V \curvearrowright \square + E_2}{V + E_2} \dots \rangle_k & \langle \frac{V \curvearrowright E_1 + \square}{E_1 + V} \dots \rangle_k & \langle \frac{V \curvearrowright \text{if } (\square) S}{\text{if } (V) S} \dots \rangle_k
 \end{array}$$

Here, E_1 , E_2 , and E represent unevaluated expressions and V represents an evaluated expression (i.e., a value). While these are the rules generated by K-Maude, in the theory of \mathbb{K} they can apply anywhere (not just at the top of the k cell). There are additional annotations for specifying more particular evaluation strategies, and can be found in documentation on \mathbb{K} [134]. We also give names to certain contexts that are evaluated differently. For example, the left-hand side (LHS) of an assignment is evaluated differently than the right-hand side (RHS). The use of this is described in Section 4.2.4.

For the remainder of this dissertation, we use the following convention with respect to the types of variables—the names given below, and common variations such as X' for X shall be given the following types, unless other types are specified: X and F have type `Id`; L has type `Location`; T has type `Type`; S has type `Struct`; E has type `Expression`; I has type `Integer`; N , B , and O have type `Natural`; C has type `Configuration` (i.e., `Bag`); M has type `Map`.

At this point, we have only looked at the very basics of \mathbb{K} , but enough to understand the semantics presented in this dissertation. For readers interested in more detail, a tremendous amount can be found in Roşu and Şerbănuţă [134], in addition to other sources [133, 146, 148]. Everything that was explained above, as well as a number of other important features have been precisely defined in those sources.

Chapter 4

Positive Semantics

This chapter describes our positive semantics of C, which include all the components necessary to describe the behavior of correct (defined) programs. We describe components of the semantics, how we tested it, and applications that can be derived from it. Much of the work in this section is from Ellison and Roşu [48] and Ellison and Roşu [47].

4.1 Introduction

In this chapter, we present a formal semantics of C that gives meaning to all correct C programs. Rather than being an “on paper” semantics, it is executable, machine readable, and has been tested against the GCC torture tests (see Section 4.3). The semantics describes the features of the C99 standard [79], but we often cite the text from the C11 standard [81]. We use the C11 text because it has superseded the C99 standard, and because it offers clearer wording and more explicit descriptions of certain kinds of behavior. To a great extent, the semantics we offer is compatible with the C11 semantics; we do not target it because it had not yet been completed during the majority of our development.

Our semantics can be considered a *freestanding* implementation of C99. The standard defines a freestanding implementation as a version of C that includes every language feature except for `_Complex` and `_Imaginary` types, and that includes only a subset of the standard library. We additionally provide a number of functions found in `math.h`, `stdio.h`, `stdlib.h`, and `string.h`, including `malloc()` and `longjmp()`.

Above all else, our semantics has been motivated by the desire to develop formal, yet practical tools. Our semantics was developed in such a way that the single definition could be used immediately for interpreting, debugging, or analysis (described in Section 4.4). At the same time, this practicality does not

mean that our definition is not formal. Being written in \mathbb{K} , a front-end of RL, it comes with a complete proof system and initial model semantics [104]. Briefly, a rewrite system is a set of rules over terms constructed from a signature. The rewrite rules match and apply everywhere, making RL a simple, uniform, and general formal computational paradigm. This is explained in greater detail in Section 3.2.

Our C semantics defines 150 C syntactic operators. The definitions of these operators are given by 1,163 semantic rules spread over 5,884 source lines of code (SLOC). However, it takes only 77 of those rules (536 SLOC) to cover the behavior of statements, and another 163 for expressions (748 SLOC). There are 505 rules for dealing with declarations and types, 115 rules for memory, and 189 technical rules defining helper operators. Finally, there are 114 rules for the core of our standard library.

4.2 The Semantics of C in \mathbb{K}

In this section, we describe the different components of our definition and give a number of example rules from the semantics. The rules found in this section are sometimes slightly simplified to focus on the semantics of correct programs. Techniques related to detecting undefined programs can be found in the next chapter. The complete semantics can be found in Appendix A, though the rules there come with less explanatory text than the rules found in this section.

4.2.1 Syntax

We use the FrontC parser, with additions made and included in CIL [114], an “off-the-shelf” C parser and transformation tool. FrontC itself parses only ANSI C (C90), but CIL extended it with syntax for C99. We use *only* the parser here, and none of the transformations of CIL; we give semantics directly to the abstract syntax tree generated by the parser. The FrontC parser (with C99 extensions) is used by a number of other tools, including CompCert [13] and Frama-C [34].

We also made our own changes to the parser. First, we made it stricter, in that it no longer accepts certain deprecated features of C like implicit `int` declarators. We additionally added syntactic support for the new fea-

tures of C11, including `_Noreturn`, `_Thread_local`, and `_Atomic(type-name)` (though we do not yet give these constructs semantics; see Section 6.1). Finally, we added our own `#pragma` for declaring LTL predicates (see Section 4.4.2).

We do not define any part of the C preprocessor (C`pp`), even though it is specified in the standard. The C`pp` is essentially a second language on top of C used for static transformation of code, including code generation and conditional compilation—it is the language of `#include` and `#define`. Instead, we simply use an off-the-shelf preprocessor (`gcc -E`), which takes C containing C`pp` directives and outputs pure C.

4.2.2 Configuration (Program + State)

The configuration of a running program is represented by nested multisets of labeled cells, and Figure 3.1 shows the most important cells used in our semantics. While this figure only shows 17 cells, we use over 90 in the full semantics. The entire configuration is shown in Section A.2. The outer `T` cell contains the cells used during program evaluation: at the top, a `k` cell contains the current computation itself and a `local` cell holds a number of cells related to control flow, and below, there are a number of cells dealing with global information.

In the `local` cell, there is a `callstack` used for calling and returning from functions, and a `control` cell which gets pushed onto the call stack. Inside the `control` cell, there is a local variable environment (`env`), a local type environment (`types`), local aggregate definitions (`structs`), a loop stack, a record of the locations that have been written to since the last sequence point (Section 5.3.2), 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 `local` cell are a number of global mappings, such as the global variable environment (`genv`), the global type environment (`gtypes`), global aggregate definitions (`gstructs`), the heap (`mem`), the dynamic allocation map (`malloced`), and a map from function-name/label pairs to continuations (for use by `goto` and `switch`).

4.2.3 Memory Layout

Our memory is essentially a map from locations to blocks of bytes. It is based on the memory model of both Blazy and Leroy [13] and Roşu et al. [138] 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.

Below we see a snippet of a memory cell, holding four bytes:

$$\langle \dots \langle \dots \langle 17 \rangle_{\text{basePtr}} \langle 0 \mapsto 7, 1 \mapsto 23, 2 \mapsto 140, 3 \mapsto 4 \rangle_{\text{bytes}} \dots \rangle_{\text{object}} \dots \rangle_{\text{mem}}$$

This says that at symbolic location 17, there is an object whose size is 4 bytes; those bytes are 7, 23, 140, and 4. All objects are broken into individual bytes, including aggregate types like arrays or `structs`, as well as base types like integers.

Our pointers are actually base/offset pairs, which we write as $\text{loc}(B, O)$, where B corresponds to the base address of an object itself, while the O represents the offset of a particular byte in the object. The base is symbolic—despite representing a location, it is not appropriate to, e.g., directly compare $B < B'$ (Section 5.3.3). It is better to think of the 17 above as representing “object 17”, as opposed to “location 17”.

When looked up, the bytes are interpreted depending on the type of the construct used to give the address. The simplest example possible is dereferencing a pointer $\text{loc}(17, 2)$ of type `unsigned char*`, which would simply yield the value 140 of type `unsigned char`. Looking up data using different pointer types requires taking into account a number of implementation-defined details such as the use of signed magnitude, one’s, or two’s complement representation, or the order of bytes (endianness). These choices are made parametric in the semantics, and can be configured depending on which implementation a user is interested in working with (Section 4.2.7).

When new objects (`ints`, 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 might seem that it would be more consistent to treat memory as

mappings from bit locations to individual bits, bitfields themselves are not addressable in C [81, §6.5.3.2:1], so we decided on this hybrid approach.

4.2.4 Basic Semantics

We now give the flavor of our semantics by examining a few of the 1,163 rules. For the rules below, recall that in \mathbb{K} what is above the line is considered the LHS of the rule, while what is below the line is considered the RHS. Parts of a rule without a line at all are considered to be on both sides of the rule. Furthermore, because we are only focusing on positive semantics in this chapter, these rules only cover the cases where a program is defined—they do not take into consideration the different ways a program could be undefined. However, such rules can be found in Section 5.3 in the next chapter.

Lookup and Assignment

We first consider one of the most basic expressions—the identifier. According to the standard, “An identifier is a primary expression, [...] designating an object (in which case it is an lvalue) or a function (in which case it is a function designator)” [81, §6.5.1:2]. Although in informal language an “lvalue” is an expression that appears on the LHS of an assignment, this is not the case according to the C standard. An lvalue can be more accurately thought of as any expression that designates a place in memory; a footnote in the standard suggests it might better be called a “locator value” [81, §6.3.2.1:1]. We denote lvalues with brackets; an lvalue that points to location L which is of type T is denoted by $[L]:T$. With this in mind, here then is our lookup rule:

$$\text{(LOOKUP)} \quad \frac{\langle X \dots \rangle_k \langle \dots X \mapsto L \dots \rangle_{\text{env}} \langle \dots X \mapsto T \dots \rangle_{\text{types}}}{[L]:T}$$

This rule is actually very similar to the example address-of rule we gave in Section 3.2. It says that when the next thing to evaluate is the program variable X , both its location L and its type T should be looked up (in the `env` and `types` cells), and the variable should be replaced by an lvalue containing those two pieces of information. We distinguish between objects and functions based on type.

In almost all contexts, this lvalue will actually get converted to the value at that location:

Except when it is the operand of the `sizeof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue) [79, §6.3.2.1:2].

We call these contexts “reval”, for “right” evaluation. Here is the rule for simplifying lvalues in the “right value” context:

$$\frac{\text{reval}([L] : T)}{\text{read}(L, T)} \quad \text{when } \neg(\text{isArrayType}(T) \vee \text{isFunctionType}(T))$$

The rule for “read” then does the actual read from memory. Its evaluation involves a series of rules whose job is to determine the size of the type, pull the right bytes from memory, and to piece them together in the right order to reconstruct the value. There are over 10 highly technical rules defining “read”, just for integer types alone. This process results in a normal value, instead of an lvalue, which we represent simply as $V : T$.

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 usually evaluated to their location as pointers. The corresponding paragraph for this array conversion is:

Except when it is the operand of the `sizeof` operator or the unary `&` operator, or is a string literal used to initialize an array, an expression that has type “array of type” is converted to an expression with type “pointer to type” that points to the initial element of the array object and is not an lvalue. [79, §6.3.2.1:3]

It is because of this difference that arrays cannot be assigned, passed as values, or returned. In our semantics, we handle it with the following rule for reval contexts:

$$\frac{\text{reval}([L] : T)}{L : \text{pointerType}(\text{innerType}(T))} \quad \text{when } \text{isArrayType}(T)$$

To handle the contexts included by this paragraph but not the first (e.g., the left hand side of assignment operators), we invent an additional context called “peval” for pointer evaluation. The rule for peval behaves the identically to reval for arrays, so we do not repeat it here. The peval rule for non-array types is to simply not change its arguments:

$$\frac{\text{peval}([L]: T) \quad \text{when } \neg(\text{isArrayType}(T) \vee \text{isFunctionType}(T))}{[L]: T}$$

Reference and Dereference

We can now take a look at the rule for the `&` operator:

$$(\text{REF}) \quad \frac{\langle \quad \&([L]: T) \quad \dots \rangle_k}{L: \text{pointerType}(T)}$$

This rule says that when the next computation to be performed is taking the address of an lvalue, it should simply be converted into a “true value” holding the same address, but whose type is a pointer type to the original type. We can expect to find an lvalue as the argument because the “reval” context does not include the arguments of the address operator.

The rule for dereference is similarly simple:

$$(\text{DEREF}) \quad \frac{\langle \quad *(L: \text{pointerType}(T)) \quad \dots \rangle_k}{[L]: T}$$

This will turn the non-lvalue into an lvalue of the same location. As with lookup, no memory is read by default. To see why, consider the expression `*x = y`; where we do not actually want to read the memory at `x`). For different options of extending this rule to catch undefined behaviors, see Section 5.3.1.

Structure Members

The standard says, “A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue” [81, §6.5.2.3:3].

Here is the rule for when the first expression is an lvalue:

$$\text{(LVALUE-DOT)} \quad \frac{\langle ([L] : \text{structType}(S)).F \ \dots \rangle_k \ \langle \dots \ S \mapsto (F \mapsto (\text{Offset}, T) \text{---}) \ \dots \rangle_{\text{structs}}}{[L + \text{Offset}] : T}$$

This rule finds the offset Offset and type T of the field F in struct S and simply adds the offset to the base address L of the struct to evaluate the expression. The result is another lvalue of the type of the field. In contrast, the rule for when the first expression is not an lvalue cannot simply work with pointers:

$$\text{(RVALUE-DOT)} \quad \frac{\langle (V : \text{structType}(S)).F \ \dots \rangle_k \ \langle \dots \ S \mapsto (F \mapsto SD \text{---}) \ \dots \rangle_{\text{structs}}}{\text{extractField}(V, SD, S, F)}$$

One situation in which this arises is when a function returns a struct, and the programmer uses the function call to access a particular field, as in the expression `fun().field`. The call to `fun()` will result in a struct value, represented in the rule above by $V : \text{structType}(S)$. The helper function `extractField` will look at the bytes of the struct (represented by V) and “read” a value of the appropriate type (SD contains the offset and type of the field). There are many rules shared by the `extractField` and `read` helpers, since both have to piece together bytes in implementation-defined orders to make new values.

The semantics for the arrow operator (`p->f`) is identical to that of the dot operator above after dereferencing the first subexpression:

$$\text{(ARROW)} \quad E \text{ -> } F \Rightarrow (*E).F$$

There are similar rules as above for union, where all offsets of a union’s fields are 0.

Multiplication (and Related Conversions)

As mentioned in Section 1.2, the rules for arithmetic in C are nontrivial. To show this in more detail, here we give many of the rules related to integer

multiplication. Here is the core multiplication rule:

$$\frac{(I_1 : T) * (I_2 : T)}{\text{arithInterpret}(T, I_1 *_{Int} I_2)} \quad \text{where hasBeenPromoted}(T)$$

This rule matches when multiplying values with identical, promoted types (more on promotion shortly). It then uses a helper operator “arithInterpret” to convert the resulting product into a proper value:

$$\frac{\text{arithInterpret}(T, I)}{I : T} \quad \text{when } \min(T) \leq I \wedge \max(T) \geq I$$

$$\frac{\text{arithInterpret}(T, I)}{\text{arithInterpret}(T, I -_{Int} (\max(T) +_{Int} 1))} \quad \text{when } I > \max(T)$$

$$\frac{\text{arithInterpret}(T, I)}{\text{arithInterpret}(T, I +_{Int} (\max(T) +_{Int} 1))} \quad \text{when } I < \min(T)$$

The first rule creates a value as long as the product is the range of the type. The next two rules collapse out-of-range products into range [81, §6.3.1.3:2].

With the above rules defined, the question becomes how to promote and convert the types of the operands so that the core multiplication rule can take effect. First, all arithmetic in C takes place at or above the size of `ints`. This means smaller types need to be coerced into `int` or `unsigned int`.

$$\langle (_ : \frac{T}{\text{promote}(T)}) * _ \dots \rangle_k \quad \text{when } \neg \text{hasBeenPromoted}(T)$$

The above rule (and its commutative partner) cause unpromoted multiplication operands to be promoted. Of the actual promotion, the standard says, “If an `int` can represent all values of the original type [...], the value is converted to an `int`; otherwise, it is converted to an `unsigned int`” [81,

§6.3.1.1:2]:

$$\frac{\text{promote}(T) \quad \text{when } \min(\mathbf{int}) \leq \min(T) \wedge \max(\mathbf{int}) \geq \max(T)}{\mathbf{int}}$$

$$\frac{\text{promote}(T) \quad \text{when } \neg(\min(\mathbf{int}) \leq \min(T) \wedge \max(\mathbf{int}) \geq \max(T))}{\mathbf{unsigned int}}$$

Finally, in order to perform the multiplication, the types of the operands have to be identical. If the types are not identical, an implicit conversion takes place to convert the different types to a common type. There are eight rules for this given in the standard. To give an idea of their flavor, we give a few of the rules for integer conversions here. First, the rule to enable conversion:

$$\left\langle \frac{I_1 : T}{\text{cast}(\tau, I_1 : T)} * \frac{I_2 : T'}{\text{cast}(\tau, I_2 : T')} \cdots \right\rangle_k \quad \text{when} \quad \begin{array}{l} T \neq T' \\ \wedge \tau = \text{arithConv}(T, T') \end{array}$$

The standard says, “if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank” [81, §6.3.1.8:1]:

$$\frac{\text{arithConv}(T, T') \quad \text{when } \text{hasSameSignedness}(T, T')}{\text{maxType}(T, T')}$$

Rank is a partial ordering on integer types based on their ranges and signedness, e.g., $\text{rank}(\mathbf{short int}) < \text{rank}(\mathbf{int})$. Additionally, the ranks of unsigned integer types equal the ranks of the corresponding signed integer types [81, §6.3.1.1:1]. Continuing with the conversion rules, “Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type” [81, §6.3.1.8:1]:

$$\frac{\left\langle \text{arithConv}(T, T') \cdots \right\rangle_k}{T} \quad \text{when} \quad \begin{array}{l} \text{isUnsigned}(T) \\ \wedge \text{isSigned}(T') \\ \wedge \text{rank}(T) \geq \text{rank}(T') \end{array}$$

and similarly for the commutative case.

The above equations use a number of helper operators in their side conditions—the definitions for “min” and “max” are given in Section 4.2.7; the other operators are defined as expected.

Malloc and Free

Here we show our semantics of `malloc` and `free`. These are functions from the standard C library that perform dynamic memory allocation and deallocation. The declarations of these functions are:

```
void *malloc(size_t size);
void free(void *ptr);
```

where `size_t` is an unsigned integer type that is implementation defined. When a programmer calls `malloc()`, an implementation can return a new pointer pointing to a new block of memory the size specified by the programmer, or it can return `NULL` (e.g., if there is no memory available).

Here is the rule for a successful call to `malloc`:

$$\begin{array}{c}
 \text{(MALLOC)} \\
 \frac{\langle \frac{\text{malloc}(N:\text{size_t})}{\text{alloc}(L, N) \curvearrowright L:\text{pointerType}(\text{void})} \cdots \rangle_k \langle \cdots \frac{\cdot}{L \mapsto N} \cdots \rangle_{\text{malloced}}}{\text{when } L \text{ is fresh}}
 \end{array}$$

If the user requests N bytes, the semantics will schedule that many bytes to be allocated at a new location and record that this memory was dynamically allocated in the `malloced` cell. Here is the related rule for a failed call to `malloc`:

$$\begin{array}{c}
 \text{(MALLOC-FAIL)} \\
 \frac{\langle \frac{\text{malloc}(\text{---})}{\text{NullPointer}:\text{pointerType}(\text{void})} \cdots \rangle_k}{}
 \end{array}$$

This rule is usually only useful when searching the state space.

A call to `free` is meant to deallocate space allocated by `malloc`. Its rule is also straightforward:

$$\begin{array}{c}
 \text{(FREE)} \\
 \frac{\langle \frac{\text{free}(L)}{\cdot} \cdots \rangle_k \langle \cdots \frac{L \mapsto N}{\cdot} \cdots \rangle_{\text{malloced}} \langle \cdots \langle \frac{L}{\cdot} \rangle_{\text{basePtr}} \cdots \rangle_{\text{object}} \cdots \rangle_{\text{mem}}}{\cdot}
 \end{array}$$

When the user wants to free a pointer L , it is removed from both the `malloced` and `mem` cells. By matching these cells, the rule ensures that the pointer has not already been freed, and once applied, ensures no other rules that use that address can match into the memory.

Setjmp and Longjmp

Finally, we show our semantics of `setjmp` and `longjmp`. These are functions from the standard C library that perform complex control flow. They are reminiscent of `call/cc`, and are often used as a kind of exception handling mechanism in C. The declarations of these functions are:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

where `jmp_buf` is an array type “suitable for holding the information needed to restore a calling environment.” A call to `setjmp` “saves its calling environment [...] for later use by the `longjmp` function.” Additionally, the call to `setjmp` evaluates to zero [81, §7.13.1]. Here is our rule for `setjmp`:

$$(\text{SETJMP}) \quad \frac{\langle \text{setjmp}(L : \text{jmp_buf}) \quad \curvearrowright \kappa \rangle_k \langle C \rangle_{\text{local}}}{\text{write}(L, C \langle \kappa \rangle_k) \quad \curvearrowright 0 : \text{int}}$$

Because `jmp_buf` is an array type, it will evaluate to an address L . In the rule above, we match the remaining computation κ (similar to a continuation), as well as the local execution environment C . This includes cells like the call stack and the map from variables to locations (which we also call the environment). The rule then causes this information to be written at the location of the `jmp_buf`.

A call to `longjmp` “restores the environment saved by the most recent invocation of [`setjmp`] with the corresponding `jmp_buf` argument” [81, §7.13.2]. When the user calls `longjmp`, this address is read to find that previous context:

$$(\text{LONGJMP}) \quad \frac{\langle \text{longjmp} \quad (\quad \frac{L : T}{\text{read}(L, T)} \quad , -) \cdots \rangle_k}{\text{longjmp-aux}}$$

and it is then restored:

$$\text{(LONGJMP-AUX)} \quad \frac{\langle \text{longjmp-aux}((C \langle \kappa \rangle_k : _), I : \text{int}) \rangle_{\text{local}}}{(\text{if } I = 0 \text{ then } 1 \text{ else } I \text{ fi}) : \text{int}} \rightsquigarrow \frac{_}{\kappa} \rangle_k \langle _ \rangle_{\text{local}} \overline{C}$$

This function returns the `val` that the user passes, unless this is a 0, in which case it returns 1.

It should be clear that these rules operate on the configuration itself, treating it as a first-class term of the formalism. The fact that \mathbb{K} allows one to grab the continuation κ as a term is what makes the semantics of these constructs so easy to define. This is in sharp opposition to semantic formalisms like SOS [128] where the context is a derivation tree and not directly accessible as an object inside a definition.

4.2.5 Static Semantics

Although our focus is on the dynamic semantics of C , some aspects of the dynamic semantics involve types. In particular, the `sizeof` construct, when applied to an expression, requires that the type of that expression be known in order to calculate its size. The meaning of initializers also involves the calculation of the type of an expression. While these types could be computed statically, we compute them dynamically as needed.

Whereas in the dynamic semantics, the majority of the action happens at the top of the `k` cell, the same is true of the `type` cell in the static semantics. We use a different cell because although we need a computational cell, it should not behave in the same way as for the dynamic semantics. For example, if we are to type a variable, we do not want this action to involve reading memory, despite the fact that the dynamic semantics would have it read memory. Not using the `k` cell is enough to prevent such rules of the dynamic semantics from applying. Moreover, the static evaluation of operators require different strictnesses than the dynamic evaluation. For example, in evaluating the ternary condition expression `(_?_:_)` in the dynamic semantics, only the first argument should be evaluated before the branch is chosen. However, in the static semantics, both branches need to be evaluated, as the types of both are needed in order to determine a common type for the result [81, §6.5.15:5–6]. Again, using a different cell for evaluation enables these changes.

The static semantics contains rules for giving types to each C expression. As with the dynamic semantics, we now give a few example rules. For variables, it has a lookup rule:

$$\text{(STATIC-LOOKUP)} \quad \frac{\langle X \dots \rangle_{\text{type}} \quad \langle \dots X \mapsto T \dots \rangle_{\text{types}}}{T}$$

that simply looks up the declared type of a variable in a map. There are also rules for literal values such as strings:

$$\text{(STATIC-STRING)} \quad \left\langle \frac{S}{\text{arrayType}(\text{char}, \text{lengthString}(S) +_{\text{Int}} 1)} \dots \right\rangle_{\text{type}}$$

The additional element in the array type is for the null terminator ('0') at the end of a string literal. For most arithmetic operators, the semantics has a rule similar to the following rule for multiplication:

$$\text{(TYPE-MULT)} \quad \left\langle \frac{T * T'}{\text{arithConv}(T, T')} \dots \right\rangle_{\text{type}}$$

The standard says of multiplication (and other similar arithmetic operators) that the “usual arithmetic conversions are performed on the operands” [81, §6.5.5]. The `arithConv` operator determines a common type for the result based on the types of the arguments and is partially defined in Section 4.2.4. Other binary operators, such as bitwise shifts, behave slightly differently. The standard says of bitwise shifts, “The type of the result is that of the promoted left operand” [81, §6.5.7:3]. In the semantics, it looks like this:

$$\text{(TYPE-LSHIFT)} \quad \left\langle \frac{T \ll _}{\text{promote}(T)} \dots \right\rangle_{\text{type}}$$

For these rules to apply, the operands of the expressions need to be first evaluated to a type. Using strictness annotations (Section 3.2) to indicate which operands are to be first evaluated, these arguments are pulled out for evaluation. After which, the rules of the static semantics (and further applications of strictness) apply recursively until a primary expression is at the top of the `type` cell, in which case the corresponding rule applies. These sub-values are then cooled, and further rules can apply to their surrounding

expressions.

4.2.6 Concurrency Semantics

Despite being used for concurrent programming, C99 as a language has no mechanisms for spawning threads. In most systems, this capability is provided by the operating system through external calls, such as in POSIX [76]. Although we focus on the semantics of C99 in this dissertation, we decided to add some of the new concurrency extensions of the newer C11 standard. We add enough detail to show that the remaining C11 concurrency features could be supported with minimal effort.

Concurrency Primitives

We handle thread creation (`thrd_create`) and joining (`thrd_join`) as well as mutex locking (`mtx_lock`) and unlocking (`mtx_unlock`). Although there are a number of varieties of locks allowed by the C11 standard, we only give semantics to the simplest variety. The above are the only functions we support from the `threads.h` header. Other concurrency features that we do not cover include `_Atomic` types, explicit memory order synchronization operations, or `_Thread_local` storage.

Spawning a new thread (using `thrd_create`) is relatively simple. The signature for this function is:

```
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

The type `thrd_t` is the type of thread identifiers, the type `thread_start_t` is the same as type `int (*)(void*)` (a function pointer taking a void-pointer and returning an `int`). According to the standard, the “`thrd_create` function creates a new thread executing `func(arg)`,” and also “sets the object pointed to by `thr` to the identifier of the newly created thread” [81, §7.26.5.1:2]. To handle this in \mathbb{K} , we first find the next available thread id and assign it at the location given as a first argument:

(THR-CREATE)

$$\langle \frac{\text{thrd_create}(L, F, V)}{*L := (Id : \text{int}); \curvearrowright \text{thrd_create}'(Id, F, V)} \cdots \rangle_k \langle \frac{Id}{Id + \text{Int } 1} \rangle_{\text{nextThreadId}}$$

This delegates the rest of the thread creation to a helper `thrd_create'`. This helper will do the actual spawning of the thread. First, it always succeeds (in our semantics), so it always returns `thrd_success`, an `enum` constant that, “is returned by a function to indicate that the requested operation succeeded” [81, §7.26.1:5]. It also updates the `threadStatus` map to say that the new thread `Id` is running. Most importantly, it creates a new thread with the proper starting cells (in particular the right environment and translation unit), and enforces the first action of that new thread to call `F` with the argument `A`:

$$\begin{array}{c}
 \text{(THRD-CREATE')} \\
 \left(\frac{\langle \dots \langle \text{thrd_create}'(Id, F, V) \dots \rangle_k \langle Tu \rangle_{\text{currTU}} \dots \rangle_{\text{thread}}}{\text{thrd_success}} \right. \\
 \left. \frac{\langle Env \rangle_{\text{genV}} \langle \frac{Status}{Status [\text{threadRunning} / Id]} \rangle_{\text{threadStatus}}}{\cdot} \right. \\
 \left. \frac{}{\langle \dots \langle F(A) \rangle_k \langle Tu \rangle_{\text{currTU}} \langle Id \rangle_{\text{threadId}} \langle Env \rangle_{\text{env}} \dots \rangle_{\text{thread}}} \right)
 \end{array}$$

Joining threads is an even more straightforward procedure. The signature for the `thrd_join` function is:

```
int thrd_join(thrd_t thr, int *res);
```

This function blocks until the thread identified by `thr` terminates, at which point it stores `thr`'s return value at `res` and returns. As a special case, if `res` is `NULL`, then `thrd_join` simply returns when `thr` exits [81, §7.26.5.6:2]. Below we give the main case, as the `NULL` case is a straightforward variation.

$$\begin{array}{c}
 \text{(THRD-JOIN)} \\
 \langle \frac{\text{thrd_join}(Id, L)}{\cdot} \dots \rangle_k \langle \dots \langle V \rangle_k \langle Id \rangle_{\text{threadId}} \dots \rangle_{\text{thread}} \\
 *L := (V : \text{int}); \curvearrowright \text{thrd_success} \\
 \text{when } L \neq \text{NULL}
 \end{array}$$

The above rule causes a call to `thrd_join` to block until the thread it is waiting for has finished evaluating to a result `V`, in which case it stores `V` at location `L` and returns `thrd_success`.

The locking and unlocking operations we define also come from C11. The operations are handled by two similar functions:

```
int mtx_lock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
```

The lock operation records a mutex pointed to by the argument as being registered in the locking thread:

$$\begin{array}{c}
 \text{(MTX-LOCK)} \\
 \frac{\langle \underline{\text{mtx_lock}(Loc:—)} \dots \rangle_k \langle B \cdot \rangle_{\text{glocks}} \langle \dots \cdot \dots \rangle_{\text{locks}}}{\text{thrd_success} \quad \quad \quad \underline{Loc} \quad \quad \quad \underline{Loc}} \quad \text{when } Loc \notin B
 \end{array}$$

while the unlock operation simply reverses this process:

$$\begin{array}{c}
 \text{(MTX-UNLOCK)} \\
 \frac{\langle \underline{\text{mtx_unlock}(Loc:—)} \dots \rangle_k \langle \dots \underline{Loc} \dots \rangle_{\text{glocks}} \langle \dots \underline{Loc} \dots \rangle_{\text{locks}}}{\text{thrd_success} \quad \quad \quad \cdot \quad \quad \quad \cdot}
 \end{array}$$

The `glocks` cell is a shared, global bag of locations, while the `locks` cell is local to the thread. The two cells together ensure that a thread can only lock locations that are not locked by any thread, and that they can only unlock locations locked by themselves.

When first formalizing the semantics of C, we did not plan to introduce concurrency. Despite that, as hoped for, nearly all rules could be left unchanged upon adding thread cells. Program termination had to be adjusted, and a new rule for thread termination had to be added. Other than these small changes, no other rules of the semantics had to be adjusted.

Concurrent Memory Model

Expressing concurrency in \mathbb{K} is relatively straightforward and has been done for a number of languages already [146]. However, the naive semantics yields a sequentially strict memory model—a model that is not supported by most hardware implementations and is too strict to capture the allowed behaviors of C. Instead, C gives “relaxed” memory guarantees that allow many operations to execute out of order. In the years preceding the formalization of the new C standard, there were many papers exploring the implications of such a memory model [9, 10, 141, 151]. This too has been captured in \mathbb{K} previously [146], but only for a small language. In the previous work, the language started with a sequentially consistent memory model and was modified to match

the x86-TSO [122] relaxed memory model. In addition, the search facilities of Maude allowed them to create a race detector that works with this new memory model. We were able to replicate their findings.

The TSO memory model has been used as a model to evaluate implementations of the C11 memory model, and is now considered to be consistent with it [9, 151]. Therefore, it makes sense to incorporate it into our semantics. For demonstration purposes, we make a few simplifying assumptions. First, we assume each thread has access to its own processor, and thus each thread has its own memory write-buffer. Additionally, we assume local variables may only be read by the thread that created them (this assumption is allowed by the standard [81, §6.2.4:5]).

To do this, we add one new cell to the configuration, to keep track of pending changes to the global state. We had to change some of the existing rules for reading and writing to use this new cell, as well as the operations that act as fences, including all the concurrency primitives. These fence operations correspond to synchronization events dictated by the standard (e.g., [81, §7.26.5.1:2] for `thrd_create`). Essentially, they are no longer allowed to take place without flushing the buffer. We go into more detail about the changes to reading and writing operations below.

First, we changed the original rule for writing a byte:

$$\frac{\langle \text{writeByte}(\text{loc}(B, O), V) \dots \rangle_k \langle \dots \langle B \rangle_{\text{basePtr}} \langle \dots \frac{M}{M[V/O]} \dots \rangle_{\text{bytes}} \dots \rangle_{\text{object}}}{\cdot}$$

Here we are assigning the byte V to offset O from the object at base B . This rule looks up the object whose base pointer is B and then updates the `bytes` map with the new byte at offset O . To accommodate a relaxed memory mode, this rule was split into two rules—one that puts the byte into a buffer local to the thread:

$$\text{(WRITE-BYTE-BUFFER)} \quad \frac{\langle \text{writeByte}(Loc, V) \dots \rangle_k \langle \dots \frac{\cdot}{\text{bwrite}(Loc, V)} \dots \rangle_{\text{buffer}}}{\cdot}$$

and another that can commit buffered bytes to memory at any time:

$$\begin{array}{c}
 \text{(COMMIT-BYTE)} \\
 \frac{\langle \dots \text{bwrite}(\text{loc}(B, O), V) \rangle_{\text{buffer}} \langle \dots \langle B \rangle_{\text{basePtr}} \langle \frac{M}{M[V/O]} \rangle_{\text{bytes}} \dots \rangle_{\text{object}}}{\cdot}
 \end{array}$$

The original rule for reading a byte from memory is simple:

$$\frac{\langle \text{readByte}(\text{loc}(B, O)) \dots \rangle_{\text{k}} \langle \dots \langle B \rangle_{\text{basePtr}} \langle \dots O \mapsto V \dots \rangle_{\text{bytes}} \dots \rangle_{\text{object}}}{V : \text{no-type}}$$

Given a pointer with base B and offset O , the object at base B is looked up and the byte given at offset O is retrieved from it. As with the rule for writing bytes above, the rule for reading bytes also becomes two rules in the relaxed memory model—one for reading bytes from the buffer:

$$\begin{array}{c}
 \text{(READ-BYTE-BUFFER)} \quad \frac{\langle \text{readByte}(Loc) \dots \rangle_{\text{k}} \langle \dots \text{bwrite}(Loc, V) M \rangle_{\text{buffer}}}{V : \text{no-type}} \\
 \text{when } Loc \notin \text{locations}(M)
 \end{array}$$

and one for reading committed bytes from memory:

$$\begin{array}{c}
 \text{(READ-BYTE-MEMORY)} \\
 \frac{\langle \text{readByte}(\text{loc}(B, O)) \dots \rangle_{\text{k}} \langle \dots \langle B \rangle_{\text{basePtr}} \langle \dots O \mapsto V \dots \rangle_{\text{bytes}} \dots \rangle_{\text{object}} \langle M \rangle_{\text{buffer}}}{V : \text{no-type}} \\
 \text{when } \text{loc}(B, O) \notin \text{locations}(M)
 \end{array}$$

In Section 4.4.2 we show how our semantics can run programs under a sequentially consistent memory model, or under a relaxed memory model, and can identify program problems due to the memory model.

4.2.7 Parametric Behavior

We chose to make our definition parametric in the implementation-defined behaviors (and are not the first to do so [13, 33]). 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 module `C-SETTINGS` starts with:

$$\begin{array}{ll}
 \text{numBytes}(\text{signed-char}) \Rightarrow 1 & \text{numBytes}(\text{short-int}) \Rightarrow 2 \\
 \text{numBytes}(\text{int}) \Rightarrow 4 & \text{numBytes}(\text{long-int}) \Rightarrow 4 \\
 \text{numBytes}(\text{long-long-int}) \Rightarrow 8 & \text{numBytes}(\text{float}) \Rightarrow 4 \\
 \text{numBytes}(\text{double}) \Rightarrow 8 & \text{numBytes}(\text{long-double}) \Rightarrow 16
 \end{array}$$

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

$$\begin{array}{l}
 \text{numBits}(T) \Rightarrow \text{numBytes}(T) * \text{bitsPerByte} \text{ where } \neg \text{isBitFieldType}(T) \\
 \text{min}(\text{int}) \Rightarrow -_{Int} (2^{\text{numBits}(\text{int}) - Int 1}) \\
 \text{max}(\text{int}) \Rightarrow 2^{\text{numBits}(\text{int}) - Int 1} -_{Int} 1
 \end{array}$$

Here we use a side condition to check when a type is *not* a bitfield. Finally, the above rules are used to define how an integer I of type T is cast to an unsigned integer type T' :

$$\begin{array}{l}
 \text{cast}(T', I : T) \Rightarrow (I \%_{Int} (\text{max}(T') +_{Int} 1)) : T' \\
 \text{where } \text{isIntegerType}(T) \wedge \text{isUnsignedIntType}(T') \wedge I > \text{max}(T')
 \end{array}$$

Here we use helper predicates in our side conditions to make sure this rule only applies when casting from integer types to unsigned integer types. There are similar equations used to define other cases.

4.2.8 Expression Evaluation Strategy

The C standard allows compilers freedom in optimizing code, 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.

To correctly capture the intended evaluation orders, we use a combination of evaluation contexts and strictness annotations. The basics of these constructs

are explained in Section 3.2. The strictness annotations are easiest to use, but least expressive. They are used when an operand should simply be evaluated before the operator can be evaluated. For example, when evaluating the type of an addition expression, both arguments must first be evaluated to a type. This is expressed by saying that the “+” operator is strict in the `type` cell:

$$\text{SYNTAX } \textit{Expression} ::= K + K \quad [\text{type-strict}]$$

This says that the arguments of the addition operator should be taken out for evaluation when the addition is being evaluated in the `type` cell. For dynamic evaluation, things are trickier. As explained in Section 4.2.4, addition is a so-called rvalue context, so its arguments must turn lvalues to rvalues. This is done with two contexts:

$$\begin{aligned} \text{CONTEXT } _ + _ ((\text{HOLE} \Rightarrow \text{reval}(\text{HOLE})), _) & \quad [\text{superheat}] \\ \text{CONTEXT } _ + _ (_, (\text{HOLE} \Rightarrow \text{reval}(\text{HOLE}))) & \quad [\text{superheat}] \end{aligned}$$

These contexts say three things. First, the operands of the addition operator need to be taken out for evaluation before the addition can be evaluated. Second, when doing so, they should be wrapped with the `reval` operator. Section 4.2.4 also gives the definition of `reval`. Third, (via the `superheat` annotation) that these strictness annotations are to be counted as truly nondeterministic actions in the transition system. Without the `superheat` annotation, it is assumed that either operand could be chosen first without affecting the result. \mathbb{K} allows the semanticist to choose this in order to help abstract the state space.

Our semantics does capture the appropriate state space, as seen in Section 4.4.2.

4.2.9 Putting It All Together with `kcc`

Using a simple frontend that mimics the behavior of GCC [58], C programs are parsed and translated into a Maude term, then reduced using the rules of our formal semantics. For defined programs, this process produces indistinguishable behavior from the same C program run as native code. We call this interpreter, obtained automatically from our formal semantics, `kcc`. As we will show in Section 4.4, `kcc` is significantly more than an interpreter—in addition to simple interpretation, it is also capable of debugging, catching

undefined behaviors, state space search, and model checking. 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 4.3). It also means people with no formal background can get use out of our semantics simply by using it as they would a compiler.

The following outlines the entire process of running a program using `kcc`. As shown in the above listing, all of this happens automatically when running `kcc` and the compiled program. Further, much of this is handled automatically by the \mathbb{K} Framework itself [150].

1. `kcc`

- (a) The program is run through an off-the-shelf preprocessor (`gcc -E`)
- (b) The program is parsed to an AST;
- (c) The AST is converted to a format recognizable by \mathbb{K} ;
- (d) Multiple translation units are kept as separate subtrees;

2. `a.out`

- (a) Any available input is collected (useful for search where input cannot be given at runtime);
- (b) Environmental settings are taken into consideration, like `DEBUG` (Section 4.4.1) and `SEARCH` (Section 4.4.2);
- (c) An “eval(ast)” term is constructed containing the program’s AST;

3. Static Semantics

- (a) The “eval” term above is turned into an initial configuration, that is, an concrete instantiation of the C configuration;
- (b) All translation units are processed for their global declarations;
- (c) A resolution phase identifies the use of identifiers with their appropriate declaration (i.e., the program is linked);
- (d) A helper map for `goto` and `switch` is constructed based on the defined labels;

4. Dynamic Semantics

- (a) An appropriate call to `main` is performed depending on how it was declared;
- (b) The program is run or searched, depending on mode;
- (c) The output is collected and displayed/returned for normal executions, or an error is reported for undefined executions.

4.3 Testing the Semantics

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

4.3.1 GCC Torture Tests

As discussed in the previous section, our semantics is encapsulated inside a drop-in replacement for GCC, 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 [61] and compare its behavior to that of GCC 4.1.2, as well as the Intel C Compiler (ICC) 11.1 and Clang 3.0 r132915 (C compiler for LLVM). We ran all compilers with optimizations turned off.

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 1093 tests, of which we excluded 267 tests because they used GCC-specific extensions or builtins, they used the `_Complex` data type or certain library functions (which are not required of a freestanding implementation of C), or they were machine dependent. This left us with 826 tests. Further manual inspection revealed an additional 50 tests that were non-conforming according to the standard (mostly signed overflow or reading from uninitialized memory), bringing us to a grand total of 776 viable tests.

In order to avoid “overfitting” our semantics to the tests, we randomly extracted about 30% of the conforming tests and developed our semantics using only this small subset (and other programs discussed in Section 4.3.2). 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%). After this initial test, we began to use all of the tests to help develop our semantics; we now pass 770 (99.2%) of the 776 compliant tests.

Compiler	Torture Tests Run (of 776)	
	Count	Percent
GCC	768	99.0
ICC	771	99.4
Clang	763	98.3
kcc	770	99.2

The 776 tests represent about 23,500 SLOC, or 30 SLOC/file.

Correctness Analysis Our executable formal semantics performed nearly as well as the best compiler we tested, and better than the others. We incorporated the passing tests into our regression suite that gets run every time we commit a change. This way, upon adding features or fixing mistakes, our accuracy can only increase.

Three of the six failed tests rely on floating point accuracy problems. Two more rely on evaluating expressions inside of function declarators, as in:

```
int fun(int i, int array[i++]) { return i; }
```

which we are not handling properly. The last is a problem with the lifetime of variable length arrays.

Coverage Analysis In order to have some measure of the effectiveness of our testing, we recorded the application of every semantic rule for all of the torture tests. Out of 887 core rules (non-library, non-helper operator), the GCC torture tests exercised 805 (91%).

In addition to getting a coverage measure, this process suggests an interesting application. For example, in the GCC tests looked at above, a rule that deals with casting large values to `unsigned int` 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.

4.3.2 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 over 17,000 SLOC in our regression tests that are run when making changes to the semantics. These tests include a number of programs from the LCC [70] and CompCert [13] compilers. We also execute the “C Reference Manual” tests (also known as `cq.c`), which go through Kernighan and Ritchie [88] 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 98% (867/887 rules).

We can successfully execute Duff’s Device [45], an unstructured `switch` statement where the `cases` 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 [118]. All of these test programs, as well as our semantics, are available from our project webpage: <http://c-semantics.googlecode.com/>.

4.4 Applications

Here we describe applications of our formal semantics, which are in addition to the interpreter already mentioned. These tools are automatically derived from the semantics—changes made to the semantics immediately affect the

tools. We are permitted this luxury because we take advantage of general purpose tools available to RL theories, of which our semantics is one. Contrast this to the nearly universal strategy of writing analysis tools independently of semantics. Instead of developing a different model for each tool, a plethora of tools can be created around a single semantic definition. These tools are essentially wrappers, or views, of the semantics.

4.4.1 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. This provides the ability to step through interesting parts of execution to find out what rules of semantics are invoked in giving meaning to a program.

In the semantics, we handle this function by giving a labeled rule that causes it to evaluate to a “void” value. It is essentially equivalent to `void __debug(int i) { }`. If this function is called during execution, it starts a debugger that allows the user to inspect the current state of the program. One 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. For example:

```
int main(void){
    for (int i = 0; i < 10; i++){ __debug(i); }
    printf("done!\n");
}
```

We can run or debug the program above as follows:

```
$ kcc debug.c
$ ./a.out # run the program normally
done!
$ DEBUG=1 ./a.out # or run it in the debugger
Debug(1)> where .
```

$$\langle _ _ \text{debug}(0:\text{int}) \dots \rangle_k \langle \dots i \mapsto L \dots \rangle_{\text{env}} \dots$$

```
Debug(1)> resume .
```

$$\langle _ _ \text{debug}(1:\text{int}) \dots \rangle_k \langle \dots i \mapsto L \dots \rangle_{\text{env}} \dots$$

The user can use this to see what the value of the `__debug` argument is each time through the loop, as well as the entire state of the program when the breakpoint was reached. The state presented to the user includes all of the cells of the language (Figure 3.1). This elided state is represented by the ellipses above. In addition to the “where” and “resume” commands, there is also a “step” command to step through the application of a single semantic rule [30, §22.1].

4.4.2 State Space Search

We can also use our semantics to do both matching-based state search and explicit state model-checking with linear temporal logic (LTL). The basic examples below show how our semantics captures the appropriate expression evaluation semantics precisely.

Exploring Evaluation Order

To show our semantics captures the evaluation orders of C expressions allowed by the specification, we examine some examples from related works. The results given below are not just theoretical results from our semantics, but are actual results obtained from executing the tools provided by our semantic framework.

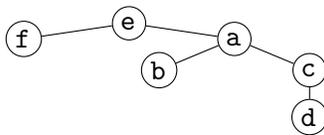
One example in the literature is given by Papaspyrou [125], which shows how C can exhibit nondeterministic behavior while staying conforming. The driving expression is the addition of two function calls. In C, function evaluation is not allowed to interleave [81, 6.5.2.2:10], so the behavior of this program is determined solely on which call happens last:

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

If `f()` is called with the argument 2 last, then the result will be 2, and similarly for 1. 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 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 for the above expression. 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 behaviors allowed by the standard:

```

$ kcc nondet.c ; SEARCH=1 ./a.out
15 solutions found
bdcafe bdcfae bdfcae bfdcae dbcafe dbcfae dbfcae dcbafe
dcbfae dcfbae dfbcae dfcbae fbdcae fdbcae fdcbae
  
```

Model Checking

In addition to the simple state search we showed above, one can also use our semantics for LTL model checking. For example, consider the following program:

```

typedef enum {green, yellow, red} state;
state lightNS = green; state lightEW = red;
int changeNS() {
    switch (lightNS) {
        case(green): lightNS = yellow; return 0;
        case(yellow): lightNS = red; return 0;
        case(red):
            if (lightEW == red) { lightNS = green; } return 0;
    }
}
...
int main(void) { while(1) { changeNS() + changeEW(); } }
  
```

This program is meant to represent two orthogonal traffic lights (`lightNS`

and `lightEW`) at the same intersection. It provides an implementation of an algorithm to change the state of the lights from green to yellow to red and back. We elide the nearly identical `changeEW()` function. The program takes advantage of the unspecified order of evaluation of addition in the expression `changeNS() + changeEW()` to nondeterministically choose the order in which the lights are changed.

There are a number of properties one might like to prove about this program, including safety and liveness properties. One safety property is that it should always be the case that at least one of the lights is red, or $\Box((\text{lightNS} == \text{red}) \vee (\text{lightEW} == \text{red}))$. We have added a special `#pragma`¹ allowing the programmer to write and name LTL formulae. If we call the above formula “safety”, then we can invoke the model checker as follows:

```
$ kcc lights.c ; MODELCHECK=safety ./a.out
result Bool: true
```

Similarly, it is important that the lights always make progress, i.e., that it is always the case the lights will eventually become green. If we try to check $\Box\Diamond(\text{lightNS} == \text{green})$, we find that it does not hold of the above program:

```
$ kcc lights.c ; MODELCHECK=progress ./a.out
result ModelCheckResult: counterexample ...
```

The reason this property is not verified is that the algorithm is wrong! Because the calls to `changeNS()` and `changeEW()` can occur in any order, it is possible for either of the lights to get stuck on red. The program starts with `ns=gre, ew=red`. Consider the following execution:

```
changeNS, changeEW => ns=yel, ew=red
changeEW, changeNS => ns=red, ew=red
changeNS, changeEW => ns=gre, ew=red
```

By alternating evaluation orders, the program can change the N/S light without ever changing the E/W light. This evaluation order is highly implausible in most C compilers, but the semantics allows it. If we fix an evaluation order by changing `changeNS() + changeEW();` to `changeNS(); changeEW();`, then the property holds:

¹A conforming way to add implementation-defined behavior to C.

```
$ kcc lights.c ; MODELCHECK=progress ./a.out
result Bool: true
```

Applying this formula to our program yields, “`result Bool: true`”, in 400 ms. If we break the algorithm by changing a `while` to an `if`, the tool instead returns a list of rules, together with the resulting states, that represent a counterexample. It is impossible to represent the entire trace here, as it is over 14 MB, but it consists of 41 function applications and instances of the if-then-else and if-then rules.

Model Checking of Concurrent C Programs

Dekker’s Algorithm We now take a look at the classical Dekker’s algorithm, in order to explore thread interleavings. The code below is based on that of Engblom [51]:

```
void dekker1(void) {
    flag1 = 1;
    turn = 2;
    while((flag2 == 1)
        && (turn == 2)) ;
    critical1();
    flag1 = 0;
}

void dekker2(void) {
    flag2 = 1;
    turn = 1;
    while((flag1 == 1)
        && (turn == 1)) ;
    critical2();
    flag2 = 0;
}
```

These two functions get called by the two threads respectively to ensure mutual exclusion of the calls to `criticaln()`. In the program we used for testing, these threads each contain infinite loops of repeated `dekker` calls, while the function `main()` waits on `thrd_join()`s. Thus, the program never terminates.

To test the mutual exclusion property, we can model check the following LTL formula: $\Box\neg(\text{enabled}(\textit{critical1}) \wedge \text{enabled}(\textit{critical2}))$, stating that the two critical sections can never be called at the same time. Applying this formula to our program yields, “`result Bool: true`”, in 400 ms. If we break the algorithm by changing a `while` to an `if`, the tool instead returns a list of rules, together with the resulting states, that represent a counterexample. It is impossible to represent the entire trace here, as it is over 14 MB, but

it consists of 41 function applications and instances of the if-then-else and if-then rules.

However, the mutual exclusion property is only guaranteed by Dekker’s algorithm for sequential memory models. By slightly modifying the semantics, as described in Section 4.2.6, we can change from a memory model with sequential consistency to a more relaxed model. Re-running the example, we see that the property no longer holds, and the tool provides a counterexample where the two critical sections can be executed concurrently. This happens because of the potential delay of updates in a truly concurrent system.

Dekker’s algorithm was investigated when the Maude model checker was introduced [46], where the authors state, “[This] algorithm example illustrates a general capability to model check in Maude *any* program (or abstraction of a program, having finitely many states) in *any* programming language: we just have to define in Maude the language’s rewriting semantics and the state predicates” [emphasis in original]. That we could use the same technique for a fully defined language lends some manner of validation to their claim.

Dining Philosophers Another classic example is the dining philosophers problem. This code is based on that of Senning [145]:

```
void philosopher(int n) {
    while(1) {
        // Hungry: obtain chopsticks
        if ( n % 2 == 0 ) { // Even number: Left, then right
            lock(&chopstick[(n+1) % NUM_PHILOSOPHERS]);
            lock(&chopstick[n]);
        } else { // Odd number: Right, then left
            lock(&chopstick[n]);
            lock(&chopstick[(n+1) % NUM_PHILOSOPHERS]);
        }
        // Eating
        // Finished Eating: release chopsticks
        unlock(&chopstick[n]);
        unlock(&chopstick[(n+1) % NUM_PHILOSOPHERS]);
    }
}
```

The above code shows a solution that has even-numbered philosophers picking up their left chopstick first, while odd-numbered philosophers pick up their right chopstick first. This strategy ensures that there is no deadlock. We can use Maude’s search command to verify there is no deadlock simply by searching for final states. Here are the results:

n	No Deadlock		With Deadlock	
	no. states	time (s)	no. states	time (s)
1	19	0.1	–	–
2	92	0.8	63	0.6
3	987	14.0	490	7.2
4	14,610	293.5	5,690	119.8
5	288,511	8,360.3	84,369	2,376.5

In the “No Deadlock” column we see the results for the code above. We were able to verify that with this algorithm, there were no deadlocks for up to five philosophers. In the “With Deadlock” column, we altered the code so that all philosophers would try to pick up their left chopstick first. Under this algorithm, we were able to find counterexamples showing that the program has deadlocks.

Chapter 5

Negative Semantics

In this chapter we discuss the negative semantics of C, that is, semantic rules to identify undefined programs. We discuss exactly what undefinedness is and what consequences it has for C, techniques to describe it formally, and evaluate our semantic rules against popular analysis tools for C on a third party benchmark and one of our own devising. Much of the work in this chapter comes from Ellison and Roşu [49] and Regehr et al. [130].

5.1 Introduction

A programming language specification or semantics has dual duty: to describe the behavior of correct programs and to identify incorrect programs. The process of identifying incorrect programs can also be seen as describing which programs do not belong to the language. Many languages come with static analyses (such as type systems) that statically exclude a variety of programs from the language, and there are rich formalisms for defining these restrictions. However, well-typed programs that “go bad” dynamically are less explored. Some languages choose to give these programs semantics involving exceptions, or similar constructs, while others choose to exclude these programs by fiat, stating that programs exhibiting such behaviors do not belong to the language. Regardless of how they are handled, semantic language definitions must specify these situations in some manner. This chapter is about these behaviors and such specifications.

In the previous chapter, we focused primarily on giving semantics to correct programs, and showed how our formal definition could yield a number of tools for exploring program evaluation. The evaluation we performed was against defined programs, and the completeness we claimed was for defined programs. In contrast, in this work we focus on identifying undefined programs. We

go into detail about what this means and how to do it, and evaluate our semantics against test suites of undefined programs.

Although there have been a number of formal semantics of various subsets of C (see Section 2.1 for an in-depth comparison), they generally focus on the semantics of correct programs only (though Norrish [119] is an exception). While it might seem that semantics will naturally capture undefined behavior simply by exclusion, because of the complexity of undefined behavior, it takes active work to avoid giving many undefined programs semantics. In addition, capturing the undefined behavior is at least as important as capturing the defined behavior, as it represents a source of many subtle program bugs. While a semantics of defined programs can be used to prove their behavioral correctness, any results are contingent upon programs actually being defined—it takes a semantics capturing undefined behavior to decide whether this is the case.

C, together with C++, is the king of undefined behavior—C has over 200 explicitly undefined categories of behavior, and more that are left implicitly undefined [81]. Many of these behaviors can not be detected statically, and as we show later (Section 5.2.6), detecting them is actually undecidable even dynamically. C is a particularly interesting case study because its undefined behaviors are truly undefined—the language has nothing to say about such programs. Moreover, the desire for fast execution combined with the acceptance of danger in the culture surrounding C means that very few implementations try to detect such errors at runtime.

Concern about undefined programs has been increasing due to the growing interest in security and safety-critical systems. However, not only are these issues broadly misunderstood by the developer community, but many tools and analyses underestimate the perniciousness of undefined behavior (see Section 5.2 for an introduction to its complexity), or even limit their input to only defined programs. For example, CompCert [94], a formally verified optimizing compiler for C, assumes its input programs are completely defined, and gives few guarantees if they contain undefined behavior. We provide this study of undefined behaviors in the hope of alleviating this obstacle to correct software.

Undefinedness tends to be considered of secondary importance in semantics, particularly because of the misconception that capturing undefined behaviors comes “for free” simply by not defining certain cases. There has been a

semantic treatment of some undefined behaviors [119], as well as a practical study of arithmetic overflow in particular [39]; both of which went to great lengths to specify undefined behavior. It can be quite difficult to cleanly separate the defined from the undefined. To see how this might be the case, consider that the negation of a context free language may not be context free, or that not all semidecidable systems are decidable. While it is true that capturing undefinedness is about not defining certain cases, this is easier said than done (see Section 5.3).

5.2 Undefinedness

In this section we examine what undefinedness is and why it is useful in C. We also look into some of the complexity and strangeness of undefined behavior. We finish with a brief overview of undefinedness in other popular languages. Other good introductions to undefinedness in C (and C++) include Regehr [129] and Lattner [92]. The fact that the best existing summaries are blog posts should indicate that there is a significant lack of academic work on undefinedness.

5.2.1 What Undefinedness Is

According to the C standard, undefined behavior is “behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements” [81, §3.4.3:1]. It goes on to say:

Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). [81, §3.4.3:2]

This effectively means that, according to the standard, undefined behavior is allowed to do anything at any time. This is discussed in more detail in Section 5.2.4. Undefined programs are *invalid* C programs, because the

standard imposes no restrictions on what they can do. Of course, particular implementations of C may guarantee particular semantics for otherwise undefined behaviors, but these are then extensions of the actual C language.

5.2.2 Undefinedness is Useful

The C standard ultimately decides which behaviors are to be undefined and which are to be defined. The most common source of undefined behaviors are behaviors that are exceptional in some way, while also hard (or impossible) to detect statically.¹ If these behaviors are undefined, an implementation of C does not need to handle them by adding complex static checks that may slow down compilation, or dynamic checks that might slow down execution of the program. This makes programs run faster.

For example, dereferencing an invalid pointer may cause a trap or fault (e.g., a Segmentation Fault); more importantly, it does not do the same thing on every platform. If the language required that all platforms behave identically, for example by throwing an exception when dereferencing an invalid pointer, a C compiler would have to generate more complicated code for dereference. It would have to generate something like: for dereference of a pointer p , if p is a valid pointer, go ahead and dereference p ; otherwise, throw an exception. This additional condition would mean slower code, which is something that the designers of C try to avoid: two of the design principles of C are that it should be “[made] fast, even if it is not guaranteed to be portable”, and that implementations should “trust the programmer” [80].

To keep the language fast, the standard states that dereferencing an invalid pointer is undefined [81, §6.5.3.3:4]. This means programs are allowed to exhibit any behavior whatsoever when they dereference an invalid pointer. However, it also means that programmers now need to worry about it, if they are interested in writing portable code. The upshot of liberal use of undefined behavior is that no runtime error checking needs to be provided by the language. This leads to the fastest possible generated code, but the tradeoff is that fewer programs are portable.

¹There are also undefined behaviors that are not hard to detect statically, such as “If two identifiers differ only in nonsignificant characters, the behavior is undefined” [81, §6.4.2:6], but are there for historical reasons or to make a compiler’s job easier.

5.2.3 Undefinedness is also a Problem

Even though undefined behavior comes with benefits, it also comes with problems. It can often confuse programmers who upon writing an undefined program, think that a compiler will generate “reasonable” behavior. In fact, compilers do many unexpected things when processing undefined programs.

For example, in the previous section we mentioned that dereferencing invalid pointers is undefined. When given this code:

```
int main(void){
    *(char*)NULL;
    return 0;
}
```

GCC,² Clang,³ and ICC⁴ will *not* generate code that segfaults, because they simply ignore the dereference of `NULL`. They are allowed to do this because dereferencing `NULL` is undefined—a compiler can do anything it wants to such an expression, including totally ignoring it.

Even worse is that compilers are at liberty to assume that undefined behavior will not occur. This assumption can lend itself to more strange consequences. One nice example is this piece of C code [113]:

```
int x;
...
if (x + 1 < x) { ... }
```

Programmers might think to use a construct like this in order to handle a possible arithmetic overflow. However, according to the standard, `x + 1` can never be less than `x` unless undefined behavior occurred (signed overflow is undefined [81, §6.5:5]). A compiler is allowed to assume undefined behavior never occurs—even if it does occur, it does not matter what happens. Therefore, a compiler is entirely justified in removing the branch entirely. In fact, GCC 4.1.2 does this at all optimization levels and Clang and GCC 4.4.4 do this at optimization levels above 0. Even though Clang and GCC only support two’s complement arithmetic, in which `INT_MAX + 1 == INT_MIN`, both compilers clearly take advantage of the undefinedness.

²v. 4.1.2 unless otherwise noted. All compilers on `x86_64` with `-O0` unless otherwise noted.

³v. 3.0

⁴v. 11.1

Here is another example where compilers take advantage of undefined behavior and produce unexpected results:

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

Because assignment is an expression in C that evaluates to “the value of the left operand after the assignment” [81, §6.5.16:3], this piece of code would seem to return 3. However, it is actually undefined because multiple writes to the same location must be sequenced (ordered) [81, §6.5:2], but addition is nondeterministic. GCC returns 4 for this program, because it transforms the code similar to the following:

```
int x = 0;
x = 1;
x = 2;
return x + x;
```

For defined programs, this transformation is completely behavior preserving. However, because it is undefined, the behavior can be, and in the case of GCC is, different than what most programmers expect.

5.2.4 Strangeness of C Undefinedness

In one sense, all undefined behaviors are equally bad because compiler optimizers are allowed to assume undefined behavior can never occur. This assumption means that really strange things can happen when undefined behavior *does* occur. For example, undefined behavior in one part of the code might actually affect code “that ran earlier”, because the compiler can reorder things. For example:

```

int main(void){
    int r = 0, d = 0;
    for (int i = 0; i < 5; i++) {
        printf("%d\n", i);
        r += 5 / d; // divides by zero
    }
    return r;
}

```

Even though the division by zero occurs after the `printf` lexically, it is not correct to assume that this program will “at least” print 0 to the screen. Again, this is because an undefined program can do *anything*. In practice, an optimizing compiler will notice that the expression `5 / d` is invariant to the loop and move it before the loop. Both GCC and ICC do this at optimization levels above 0. This means on a machine that faults when doing division by zero, nothing will be printed to the screen except for the fault. Again, this is correct behavior according to the C standard because the program triggers undefined behavior.

5.2.5 Implementation-Dependent Undefined Behavior

The C standard allows implementations to choose how they behave for certain kinds of behavior. So far in this chapter we have discussed only undefined behavior, for which implementations may do whatever they want. However, there are other kinds of behavior, including unspecified behavior and implementation-defined behavior, which we recall the definitions of here [81, §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.

An example of unspecified behavior is the order in which summands are evaluated in an addition. An example of implementation-defined behavior is the size of an `int`. Whether or not a program is undefined may actually depend

on the choices made for an implementation regarding implementation-defined or unspecified behaviors.

Undefinedness Depending on Implementation-Defined Behavior

Depending on choices of implementation-defined behavior, behavior can be defined or not. For example:

```
int* p = malloc(4);
if (p) { *p = 1000; }
```

In this code, if `ints` are 4 bytes long, then the above code is free from undefined behaviors. If instead, `ints` are 8 bytes long, then the above will make an undefined memory read outside the bounds of the object pointed to by `p`. In practice, this means that programmers must be intimately familiar with the implementation-defined choices of their compiler in order to avoid potential undefinedness arising from it.

Undefinedness Depending on Unspecified Behavior

Like implementation-defined undefined behavior above, undefined behavior can also depend on unspecified behavior. However, while implementation-defined behavior must be documented [81, §3.19.1] so that programmers may rely on it, unspecified behavior has no such requirement. An implementation is allowed to have different unspecified behaviors in different situations, and may even change them at runtime.

One such example is evaluation order. Because evaluation order is almost completely unspecified in C, an implementation may take advantage of undefined behavior found on only some of these orderings. For example, any implementation is allowed to “miscompile” this code:

```
int d = 5;
int setDenom(int x){
    return d = x;
}
int main(void) {
    return (10/d) + setDenom(0);
}
```

because there is an evaluation strategy (e.g., right-to-left) that would set `d` to

0 before doing the division. While GCC compiles this code and generates an executable containing no runtime error, CompCert [94], a formally verified optimizing compiler for C, generates code that exhibits a division by zero. Both of these behaviors are correct because the program contains reachable undefined behavior. In practice, this means that any tool seeking to identify all undefined behaviors must search all possible evaluation strategies.

5.2.6 Difficulties in Detecting Undefined Behavior

In general, detecting undefined behavior is undecidable even with dynamic information. Consider the following example:

```
int main(void){
    guard();
    5 / 0;
}
```

The undefinedness of this program is based on what happens in the `guard()` function. Only if one can show that `guard()` will terminate can one conclude that this program has undefined behaviors. However, showing that `guard()` terminates, even with runtime information, is undecidable.

Although it is impossible (in general) to prove that a program is free from undefined behaviors, this raises the question of whether one can *monitor* for undefined behaviors. The question is somewhat hard to pin down—as we saw in Section 5.2.3, a smart compiler may detect undefined code statically and generate target code that does not contain the same behaviors. This means a monitor or even state-space search tool would not be able to detect such undefined behavior at runtime, even though the original program contained it. If we instead assume we will monitor the code as run on an “abstract machine”, we can give more concrete answers.

First, it is both decidable and feasible to monitor an execution and detect any undefined behavior, as long as the program is deterministic. By deterministic we mean there is only a single path of execution (or all alternatives join back to the main path after a bounded number of steps). It is feasible because one could simply check the list of undefined behaviors against all the alternatives before executing any step. Because all decisions would be joinable, only a fixed amount of computation would be needed to check each step.

For nondeterministic single-threaded⁵ programs, one may need to keep arbitrary amounts of information, making the problem decidable but intractable. Consider this program:

```
int r = 0;
int flip() {
    // return 0 or 1 nondeterministically
}
int main(void){
    while(true){
        r = (r << 1) + flip();
    }
}
```

At iteration n of the loop above, r can be any one of 2^n values. Because undefinedness can depend on the particular value of a variable, all these possible states would need to be stored and checked at each step of computation by a monitor. The above argument could be reformulated to encode r using allocated memory, avoiding the limited sizes of builtin types like `int`, but the presentation would be more complicated.

If multiple threads are introduced, then the problem becomes undecidable. The reason is similar to the original argument—because there are no fairness restrictions on thread scheduling, at any point, the scheduler can decide to let a long-running thread continue running.

```
// thread 1                // thread 2
while (guard()) {}        5 / d;
d = 0;
```

In this example, if one could show that the loop must eventually terminate, then running thread 1 to completion followed by thread 2 would exhibit undefined behavior. However, showing that the loop terminates is undecidable.

5.2.7 Undefinedness in Other Languages

It should be clear at this point that undefinedness is a huge part of the C language, but other languages also have undefined behavior. The documentation or specifications of many popular languages identify undefined programs

⁵Threads were added to C in C11 [81].

that are allowed to do anything (including crash). For example, LLVM includes a number of undefined behaviors, including calling a function using the wrong calling convention [93]. Scheme’s specification describes undefined behavior in relation to `callcc` and `dynamic-wind` [87, p. 34]. Even Haskell, an otherwise safe and pure language, has undefined behavior in a number of unsafe libraries, such as `Unsafe.Coerce` and `System.IO.Unsafe`. There are many other examples of this in other programming languages, including Perl [41] and Ruby [140].⁶

Even languages without undefined behavior run into many of the same specification problems. Any language with constructs having exceptional behavior, such a division by zero, needs to be able to specify or define the behavior of these cases. These kinds of behavior are invariably of the form, “the `—` construct is defined as `—`. However, in some special case `—`, it raises an exception instead.” For example, the Java standard states,

The binary `/` operator performs division, producing the quotient of its operands [...] if the value of the divisor in an integer division is 0, then an `ArithmeticException` is thrown. [65, §15.17.2]

Similarly, the SML Basis Library standard states:

`[i div j]` returns the greatest integer less than or equal to the quotient of `i` by `j`. [...] It raises [...] `Div` when `j = 0`. [62]

This pattern comes up frequently enough in most languages that it is worthy of investigation. We investigate ways of formally specifying such behaviors in Section 5.3.

5.3 Semantics-Based Undefinedness Checking

As we explained in Section 5.2.7, most languages have some form of undefined, or at least exceptional, behavior. When formalizing such languages, this behavior needs to be formalized as well. We were faced with this problem when developing our formal semantics for C [48]. At first we believed that

⁶Though the Ruby standard uses the word “unspecified”, they define this to include behavior “not necessarily defined for any particular implementation” [140, §4.17].

detecting undefinedness using a semantics would simply be a matter of running the program using the semantics and letting it get stuck where there was no semantic rule for a behavior. We have come to realize that in fact, quite a lot of work needs to go on to enable these behaviors to be caught.

In this section we explain a number of techniques for dealing with undefinedness semantically. In doing so, we address most of the issues used as examples in Section 5.2.

5.3.1 Using Side Conditions and Checks to Limit Rules

By bolstering particular rules with side conditions, we can catch some undefined behavior. We have employed this technique in our semantics to catch much of the undefined behavior we are capable of catching.

Division by Zero

The simplest example of using side conditions to catch undefined behavior is in a division. In C, the following unconditional rule gives the semantics of integer division for correct programs:

$$\frac{\langle I / J \ \dots \rangle_k}{I /_{Int} J}$$

Of course, this rule is not good for programs that *do* divide by zero. In such a case, the rule might turn “/_{Int}” into a constructor for integers, where suddenly terms like $5 /_{Int} 0$ are introduced into the semantics. Programs like:

```
int main(void){
    5/0;
    return 0;
}
```

might actually be given complete meanings without getting stuck, because the semicolon operator throws away the value computed by its expression.

One way to solve this issue is simply by adding a side condition on the division rule requiring “ $J \neq 0$ ”. This will cause the rule to only define the defined cases, and let the semantics get stuck on the undefined case. In addition, human-readable error messages, like the one shown in Section 5.4.1,

can be obtained by inverting the side conditions preventing undefined behavior for occurring:

$$\langle \frac{I // J}{\text{reportError}(\text{"Division by zero"})} \dots \rangle_k \quad \text{when } J = 0$$

Array Length

In C, arrays must have length at least 1 [81, §6.7.6.2:1&5]. However, without taking this fact into consideration, it is easy to give semantics to arrays of any non-negative length, simply by allowing the size to be any natural number. If they would be used at runtime, the problem would be detected, but simply declaring them would slip through. We had precisely this problem in earlier versions of our semantics. To detect this problem, the semantics needs an additional constraint on top of allowing any natural—it must also be non-zero.

Dereferencing

This most basic form of the dereferencing rule (Section 4.2.4) rule says that dereferencing a location L of type pointer-to- T ($L : \text{ptrType}(T)$) yields an lvalue L of type T ($[L] : T$). This rule is completely correct according to the semantics of C [81, §6.5.3.2:4] in that it works for any defined program. However, it fails to detect undefined programs including dereferencing `void` [81, §6.3.2.1:1] or `null` [81, §6.3.2.3:3] pointers. In a program like:

```
int main(void){
    *NULL;
    return 0;
}
```

this rule would apply to `*NULL`, then the result ($[NULL] : \text{void}$) would be immediately thrown away (according to the semantics of “;”). The program would then return 0 and completely miss the fact that the program was undefined.

In order to catch these undefined behaviors, it could be rewritten as:

$$\text{(DEREF-SAFER)} \quad \langle \frac{*(L : \text{ptrType}(T)) \dots}{[L] : T} \rangle_k \quad \text{when } T \neq \text{void} \wedge L \neq \text{NULL}$$

If this is the only rule in the semantics for pointer dereferencing, then the semantics will get stuck when trying to dereference NULL or trying to dereference a void pointer.

One major downside with this technique is in making rules more complicated and more difficult to understand. For complex side conditions involving multiple parts of the state, including cells not otherwise needed by the positive rule, this is a big problem. To take pointer dereferencing again as an example, we also want to eliminate the possibility of dereferencing memory that is no longer “live”—either variables that are no longer in scope, or allocated memory that has since been **freed**. Here is the safest (and most verbose) version of the rule:

$$\begin{array}{c}
 \text{(DEREF-SAFEST)} \quad \frac{\langle *(\text{loc}(B, O) : \text{ptrType}(T)) \ \cdots \rangle_k \langle B \rangle_{\text{basePtr}} \langle Len \rangle_{\text{len}}}{[\text{loc}(B, O)] : T} \\
 \text{when } T \neq \text{void} \wedge O < Len
 \end{array}$$

The above rule now additionally checks that the location is still alive (by matching an object in the memory), and checks that the pointer is in bounds (by comparing against the length of the memory object). Locations are represented as base/offset pairs $\text{loc}(B, O)$, which is explained in detail in Section 5.3.3. The rule has become much more complicated. The beauty and simplicity of the original semantic rule has been erased, simply to catch undesirable cases.

A slight variation involves embedding the safety checks in the main computation. This is useful when the safety condition is complicated or involves other parts of the state. The above rule can be rewritten as two rules like so:

$$\text{(DEREF-SAFEST-EMBEDDED)} \quad \frac{\langle \frac{* (L : \text{ptrType}(T))}{\text{checkDeref}(L, T)} \ \cdots \rangle_k}{\text{checkDeref}(L, T) \curvearrowright [L] : T}$$

$$\begin{array}{c}
 \text{(CHECKDEREF)} \quad \frac{\langle \text{checkDeref}(\text{loc}(B, O), T) \ \cdots \rangle_k \langle B \rangle_{\text{basePtr}} \langle Len \rangle_{\text{len}}}{\cdot} \\
 \text{when } O < Len \wedge T \neq \text{void}
 \end{array}$$

Notice that checkDerefLoc is “blocking” the top of the k cell. As long as it stays there, no rules that match other constructs on the top of k can apply. If

checkDerefLoc succeeds, it will simply evaluate to the unit of the \surd construct and disappear. This is called “dissolving”.

The DEREF-SAFEST-EMBEDDED rule could be rewritten to use a side condition, but this would require passing the entire context (in particular, memory) to the helper-function as an argument. This works, but the rule becomes artificially complex.

Multiplication

In order to catch signed overflow, we need to add side conditions to the rules we gave in Section 4.2.4 for arithInterpret. Although most of the rule stays the same, we need to add a restriction to make sure no semantics is given to out-of-range signed types.

$$\frac{\text{arithInterpret}(T, I)}{\text{arithInterpret}(T, I -_{Int} (\max(T) +_{Int} 1))} \quad \text{when } \begin{array}{l} \text{isUnsignedIntType}(T) \\ \wedge I > \max(T) \end{array}$$

$$\frac{\text{arithInterpret}(T, I)}{\text{arithInterpret}(T, I +_{Int} (\max(T) +_{Int} 1))} \quad \text{when } \begin{array}{l} \text{isUnsignedIntType}(T) \\ \wedge I < \min(T) \end{array}$$

Simply by adding side conditions stating that these rules only apply to unsigned integer types, we now catch signed overflow here.

Data Races

Data races are defined in the C standard as follows:

The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. [81, §5.1.2.4:25]

We can express a write-write conflict in our semantics using the following rule for checking such a condition:

$$\frac{\langle \text{---} \langle \text{write}(L, \text{---}, Len) \cdots \rangle_{\text{thread}} \langle \text{write}(L', \text{---}, Len') \cdots \rangle_{\text{thread}} \rangle_{\text{threads}}}{\text{when } \text{overlaps}((L, Len), (L', Len'))}$$

where `write` is an operation taking a location, a value to write, and the length of the value. The rule matches two concurrently executing threads that are both writing at the same time. The side condition checks to see if the memory locations written to overlap, using the “overlaps” operation, and if so, dissolves all the threads in order to halt computation. The similar rule for write-read conflicts is nearly identical to the above rule.

In the process of executing a concurrent program, it is possible that such a rule is activated and will trigger an error message about a race being detected. However, for many programs with races, such a problem only becomes evident in a relatively small number of allowed executions. For such bugs, search through the state space is required. We show an example of using this rule with search to catch data races in Section 5.4.2. \mathbb{K} has been previously used to detect data races in Șerbănuță [146] using a similar mechanism.

5.3.2 Storing Additional Information

It is not enough to add new rules or side conditions to existing rules if the semantics does not keep track of all the pertinent data to be used in the specifications.

Unsequenced Reads and Writes

As explained in Section 5.2.3, unsequenced writes or an unsequenced write and read of the same object is undefined. This means that if there are two writes, or a write and a read to the same object that are unsequenced (i.e., either is allowed to happen before the other), then the expression is undefined. Examples of expressions made undefined by this clause include `(x=0)+(x=1)` and `(x=0)+x` and `x=x++` and `*p=x++`, for `int x` and `int* p=&x`. This relation is related to the concept of “sequence points”, also defined by the standard. Sequence points cause the expressions they fall between to be sequenced. The most common example of a sequence point is the semicolon, i.e., the end of an expression-statement. All previous evaluations and side effects must be complete before crossing sequence points.

In order to catch this in our semantics, we keep track of all the locations that have been written to since the last sequence point in a set called `locsWrittenTo` (see Figure 3.1). Whenever we write to or read from a location, we first check

this set to make sure the location had not previously been written to:

$$\frac{\langle \underline{\text{writeByte}}(Loc, V) \dots \rangle_k}{\text{writeByte}'} \quad \langle \underline{S} \cdot \rangle_{\text{locsWrittenTo}} \quad \text{when } Loc \notin S$$

$$\frac{\langle \underline{\text{readByte}}(Loc) \dots \rangle_k}{\text{readByte}'}}{\langle S \rangle_{\text{locsWrittenTo}}} \quad \text{when } Loc \notin S$$

After either of the above rules have executed, the primed operations will take care of any additional checks and eventually the actual writing or reading. Finally, when we encounter a sequence point, we empty the `locsWrittenTo` set:

$$\frac{\langle \underline{\text{seqPoint}} \dots \rangle_k}{\cdot} \quad \langle \underline{S} \rangle_{\text{locsWrittenTo}}$$

Sequence points are generated in the appropriate places [81, Appx. J] by other rules. For example, the rule for an expression statement is:

$$\frac{\langle \underline{V; \dots} \rangle_k}{\text{seqPoint}}$$

and the rule for the sequencing expression is:

$$\frac{\langle \underline{V, K} \dots \rangle_k}{\text{seqPoint} \curvearrowright K}$$

With the above rules for inserting and removing items from the `locsWrittenTo` cell in place, the rules given at the start of this section for reading and writing bytes are appropriately constrained.

These rules allow reading and writing on a particular path to check for bad behaviors, but this says nothing about whether the semantics can capture all possible paths of evaluation. For example, if the above rules were used to evaluate the expression $(x=0) + x$ right to left, no undefined behavior would be detected, despite the fact that such behavior would be detected if evaluated left to right. Therefore, it is essential that we correctly capture all paths of evaluation. This was addressed to some extent in Section 4.2.8, but we revisit it briefly here in the context of detecting undefined evaluations.

When two expressions are unsequenced, it means that evaluation can happen in any order. Thus, it is natural to map unsequenced behavior into nondeterministic behavior. This way, we can use state space exploration as a single mechanism to find unsequenced behavior. To identify this kind of undefined behavior naively can be incredibly computationally expensive; some optimizations are necessary to make this feasible. We offer two such optimizations below.

First, with a little case analysis of the definition of the sequencing relation, it is clear that there can be no sequenced write before a read of the same object with no intervening sequence point. This means that if in searching the semantic state space, we find an execution in which the write of a scalar object happens before a write or read of the same object with no intervening sequence point, then we can conclude that this write/write or write/read pair is unsequenced. Whenever a write is made, its location is recorded in the `locsWrittenTo` cell, which is emptied whenever a sequence point is crossed. This cell is first checked whenever a read or write is made to ensure that there is no conflict. This strategy has the added benefit that some undefined behaviors of this kind can be detected even during interpretation (where only a single path through the state space is explored). It is similar to the strategy used by Norrish [119].

However, the strategy we outlined above has the added advantage of being able to detect some of these undefined behaviors even during interpretation. Second, it turns out that a large subset of allowed orderings do not need to be considered in order to detect undefined behavior or possible nondeterministic behaviors. Because we are looking for writes *before* other events, we can take the liberty of applying side effects immediately instead of delaying them.

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'' reads or writes 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.

This shrinks the state space dramatically, while at the same time not missing any undefined behavior.

Const-Correctness

Another example of needing to keep additional information is specifying `const`-correctness. In C, a type can have the type qualifier `const`, meaning it is unchangeable after initialization. Writes can only occur through non-`const` types [81, §6.3.2.1:1, §6.5.16:1]. For correct programs, this modifier can be completely ignored, since it is only there to help the programmer catch mistakes. To actually catch those mistakes requires the semantics to keep track of `const` modifiers and to check them during all modifications and conversions.

One might think that it is possible to soundly and completely check for `const`-correctness statically—after all, it is generally not allowed to drop qualifiers on pointers [81, §6.3.2.3:2], meaning one cannot simply write code like this:

```
const char p[] = "hello";
char *q = (char*)p;
```

With this in mind, one could check that no `const`s are dropped in conversion and no writes occur through `const` types. However, this is not sufficient—there are ways around the conversion, such as this:

```
const char p[] = "hello";
char *q = strchr(p, p[0]); // removes const
```

The `strchr` library function

```
char *strchr(const char *s, int c);
```

returns a pointer to the first instance of `c` in `s`. By calling it with `p` and `p[0]`, this function returns a pointer to the same string, but without the `const` modifier. This is completely defined by itself, but if a write occurs through pointer `q`, the standard says that it is undefined [81, §6.7.3:6]. We handle this in our semantics by marking memory that was defined with `const` by placing these locations into a set named `notWritable`, then checking this fact

during subsequent writes:

$$\frac{\langle \text{writeByte}' (Loc, V) \dots \rangle_k \langle S \rangle_{\text{notWritable}} \quad \text{when } Loc \notin S}{\text{writeByte}''}$$

5.3.3 Symbolic Behavior

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. Symbolic execution is straightforward to achieve using a rewriting-based semantics: whether a term is concrete or abstract makes no difference to the theory. Rules designed to work with concrete terms do not need to be changed in order to work with symbolic terms.

Memory Locations

As we explained in Section 4.2.3, we treat pointers not as concrete integers, but as symbolic values. These values 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 [138]. In this context, it takes advantage of the fact that addresses of local variables and memory returned from allocation functions like `malloc()` are unspecified [81, §7.20.3]. For example, take the following program:

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

If we gave objects concrete, numerical addresses, then they would always be comparable. However, this piece of code is actually undefined according to the standard [81, §6.5.8:5]. Symbolic locations that are actually base/offset pairs allow us to detect this program as problematic. We only give semantics to relational pointer comparisons where the two addresses share a common base. Thus, evaluation gets stuck on the program above:

```

$ gcc bad_comparison.c ; ./a.out
ERROR encountered while executing this program.
Description: Cannot apply '<' to different base objects.
Function: main
Line: 3

```

Of course, sometimes locations are comparable. There are a number of guarantees 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). If we take the following code instead:

```

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

```

the addresses of `a` and `b` are guaranteed to be in order [81, §6.5.8:5], and in fact our semantics finds the comparison to be true because the pointers share a common base. The above is accomplished with rules like the following for each relational operator:

$$\frac{\langle (\text{loc}(B, O) : T) < (\text{loc}(B, O') : T) \dots \rangle_k \quad \text{when } O < O'}{1 : \text{int}}$$

$$\frac{\langle (\text{loc}(B, O) : T) < (\text{loc}(B, O') : T) \dots \rangle_k \quad \text{when } O \not< O'}{0 : \text{int}}$$

Note that these rules only apply when the bases are identical, and so compare offsets within an object.

Storing Pointers

Another example of the use of symbolic terms in our semantics is how we store pointers in memory. Because all data must be split into bytes to be stored in memory, the same must happen with pointers stored in memory. However, because our pointers are not actual numbers, they cannot be split directly; instead, we split them symbolically. Assuming a particular pointer

$\text{loc}(B, O)$ was four bytes long, it is split into the list of bytes:

```

subObject(loc(B, O), 0),
subObject(loc(B, O), 1),
subObject(loc(B, O), 2),
subObject(loc(B, O), 3)

```

where the first argument of `subObject` is the object in question and the second argument is which byte this represents. This allows the reconstruction of the original pointer, but only if given all the bytes. This program demonstrates its utility:

```

int main(void) {
    int x = 5, y = 6;
    int *p = &x, *q = &y;
    char *a = (char*)&p, *b = (char*)&q;
    a[0] = b[0]; a[1] = b[1]; a[2] = b[2];
    // *p is not defined yet
    a[3] = b[3]; // needs all bytes
    return *p; // returns 6
}

```

Any particular byte-splitting mechanism would mean over-specification—a user could take advantage of it to run code that is not necessarily defined.

Indeterminate Memory

Another example can be seen when copying a struct one byte at a time (as in a C implementation of `memcpy()`); every byte needs to be copied, even uninitialized fields (or padding), and no error should occur [81, §6.2.6.1:4]. Because of this, our semantics must give it meaning. Using concrete, perhaps arbitrary, values to represent unknowns 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 in undefined ways. We store these unknown bytes in memory as `unknown(N)` where N is the number of unknown bits.

In C99, unknown values are generally not allowed to be used under the possibility that they may produce a trap (an error) [79, §6.2.6.1:5]. Similarly,

in our semantics, such unknown bytes may not be used by most operations. However, exceptions are made when using an unsigned-character type [79, §6.2.6.1:3–4]—this special case is represented in our semantics by an additional rule allowing such an unknown value to be read by lvalues of the allowed type.

5.3.4 Suggested Semantic Styles for Undefinedness

In this section, we suggest two new specification techniques for capturing undefined or exceptional behavior based on our experience in capturing undefinedness in C. These are untested (we know of no semantic framework incorporating them), but we think they would make expressing undefined behavior much more straightforward.

Inclusion/Exclusion Rules

One nice way to specify exceptional behavior would be to define additional “negative” semantic rules to catch the special cases. For example, in addition to the Deref rule given earlier, add the following two rules:

$$(\text{Deref-NEG1}) \quad \left\langle \frac{* (L : \text{ptrType}(\text{void}))}{\text{reportError}(\text{“Cannot dereference void pointers”})} \dots \right\rangle_k$$

$$(\text{Deref-NEG2}) \quad \left\langle \frac{* (\text{NULL} : \text{ptrType}(T))}{\text{reportError}(\text{“Cannot dereference null pointers”})} \dots \right\rangle_k$$

For this definitional strategy to make sense, later rules must be applied before earlier rules. Each additional rule acts as a refinement on the previous rule. Simply having multiple rules is much cleaner than rules with side conditions—it allows the primary, unexceptional case to be emphasized because it is presented without side conditions. However, this strategy trades off the complexity of side conditions for the complexity of rule precedence.

It is possible for rule precedence to be supported by a semantic framework as syntactic sugar, where it automatically adds side conditions necessary to prevent earlier rules from executing first. It should be clear that one could hand-write these side conditions, but the whole point of this strategy is to avoid explicit side conditions in order to make the rules simpler.

It is not enough to consider exploring the transition system for these reportError states, since this mechanism is also useful for defined but exceptional behavior. In such cases, if one were to allow the rules to apply in any order and then analyze the resulting transition system, it would be difficult to identify which paths should be removed. Consider the following three rules for division:

$$\langle \frac{I / J}{I /_{Real} J} \dots \rangle_k \quad \leftarrow \quad \langle \frac{I / 0}{INFINITY} \dots \rangle_k \quad \leftarrow \quad \langle \frac{0 / 0}{NAN} \dots \rangle_k$$

These rules should be tried right to left until one matches. They are similar to the rules of IEEE-754 floating point, which evaluates division by zero to INFINITY or NAN values.

Declarative Specification

An additional possibility is to again start with only the original positive semantic rule, but then to add declarative specifications on top of that. For example, using LTL and configuration patterns, we could specify both

$$\begin{aligned} & \Box \neg \langle * (L : \text{ptrType}(\text{void})) \dots \rangle_k \\ \text{and } & \Box \neg \langle * (\text{NULL} : \text{ptrType}(T)) \dots \rangle_k \end{aligned}$$

The first property states that it is never the case that the next action to perform is dereferencing a void pointer. The second property states that it is never the case that the next action to perform is dereferencing a null pointer. Using a temporal logic to add these negative “axioms” to the semantics has the advantage of being able to capture undefined behavior that might only occur on one path in the transition system. For example, this property states that read-write data races are not allowed:

$$\Box \neg (\langle \text{read}(L, T) \dots \rangle_k \langle \text{write}(L', T', V) \dots \rangle_k) \\ \text{when overlaps}((L, T), (L', T'))$$

5.4 Applications

In this section, we describe the applications made available by our negative semantics. These include a run-time checker for undefined behavior, as well as a space-state exploration tool capable of uncovering program errors such as data races.

5.4.1 A Semantics-Based Undefinedness Checker

By using the three techniques described in Section 5.3, we improved our formal semantics of C (Chapter 4) into a tool capable of recognizing a wide range of undefined behaviors. While the original semantics was capable of catching a handful of undefined behaviors, in general each additional behavior we caught involved a reworking of at least one semantic rule.

Our tool is capable of detecting undefined behaviors simply by running them through the semantics. As described in Section 4.2.9, this is done using a wrapper, mimicking GCC, we built around the semantics. We report on the capabilities of this tool as compared to other analysis tools in Section 5.5. While `kcc` can run defined programs:

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

it can also report on undefined programs. If we take the third example in Section 5.2.3:

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

and run it in `kcc` we get:

```

ERROR encountered while executing this program.
=====
Error: 00016
Description: Unsequenced side effect on scalar
object with side effect of same object.
=====
Function: main
Line: 3

```

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

```

int main(void) {
    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 undefined, and our semantics will detect trying to read past the end of the array. Because this program has no meaning, our semantics “gets stuck” when exploring its behavior. It is through this simple mechanism that we can identify undefined programs and report them to the user. By default, when a program gets stuck, we report the state of the configuration (a concrete instance of that shown in Figure 3.1) and what exactly the semantics was trying to do at the time of the problem. We have also begun to add explicit error messages for common problems—here is the output from our tool for this code:

```

$ gcc buggy_strcpy.c ; ./a.out
ERROR encountered while executing this program.
=====
Error: 00002
Description: Reading outside the bounds of an object.
=====
Function: strcpy
Line: 3

```

Test Case Reduction

Our semantics-based undefinedness checker has already found one serious application—automatic test case reduction [130]. Test-case reduction refers to the process of taking a program that exhibits a bug and trying to find a smaller program (usually a sub-program) that exhibits the same bug. While test-case reduction could apply to the reduction of any buggy program, we have applied it to a particular class of programs that are machine generated to find bugs in compilers.

Csmith [168] is a C program test generator that generates random conforming programs from a large, expressive subset of the C language. These tests are then used to perform differential testing among C compilers to find compilation bugs. To date, the Csmith team has found more than 325 bugs in common compilers like GCC and Clang. The programs Csmith generates are almost always too large (many between 1,000 and 10,000 SLOC) to submit as bug reports and need to be reduced. The reduction process is semi-automatic, but is riddled with the possibility of introducing undefined behavior. Until now, these tests would have to be carefully examined by hand for undefined behavior, because any such behavior would render the tests invalid. Consider this program:

```

int main (void) {
    int x;
    x = 2;
    return x + 1;
}

```

Assume that compiler A emits code that properly returns 3 while compiler B is

buggy and generates code returning a different result. The goal of a test-case reducer is to create the smallest possible program triggering the bug in B. During reduction many variants will be produced, perhaps including this one where the line of code assigning a value to `x` has been removed:

```
int main (void) {
    int x;
    return x + 1;
}
```

This variant, however, is not a valid test case. Even if this variant exhibits the desired behavior—compilers A and B return different results—the divergence is potentially due to its reliance on undefined behavior: reading uninitialized storage. In fact, on a common Linux platform, GCC and Clang emit code returning different results for this variant, even when optimizations are disabled. Compiler developers are typically most unhappy to receive a bug report whose test input relies on undefined or unspecified behavior. This is not a hypothetical problem—a web page for the Keil C compiler states that “Fewer than 1% of the bug reports we receive are actually bugs.”⁷

Our recent work [130] has shown that a tool capable of identifying undefined behaviors is necessary to solve this problem, and that our tool is capable of filling the role. Our undefinedness checker is currently being used by the Csmith team and has allowed them to more completely automate the process and reduce the tests more aggressively.

No major changes were needed in `kcc` to make it useful in test-case reduction. However, we added English-language error messages for most of the common undefined behaviors, which made it easier to understand exactly how variants go wrong. Additionally, we added detectors for some previously-uncaught undefined behaviors to the tool because those behaviors were found in variants. Any errors reported by `kcc` are guaranteed to be real errors in the program, under the assumption that the underlying semantics accurately captures C. Since `kcc` focuses entirely on problems detectable at run time, it catches very few errors detectable statically. In practice, this is not an issue since the compilers we are testing are able to identify these problems on their own.

⁷<http://www.keil.com/support/bugreport.asp>

5.4.2 State Space Search Revisited

To start with a simple example from Papaspyrou and Maćoř [126], we take a look at $x+(x=1)$ in an environment where x is 0. This expression is undefined because the read of x (the lone x) is unsequenced with respect to the write of x (the assignment), as described in Section 5.2.3. Using our semantics to do a search of the behaviors of this expression finds this unsequenced read/write pair, and reports an error. Norrish [119] offers the deceptively simple 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 unsequenced assignments to x . Our semantics reports an error for this expression as well.

Another kind of example where search is usually needed is in detecting data races. Many thread interleavings hide the problem, and only an exhaustive search is guaranteed to find it. Take the following simple program to start:

```

int global;
int f(void* a){
    global = 1;
    return 0;
}
int g(void* a){
    global = 2;
    return 0;
}
int main(void) {
    thrd_t t1, t2;
    thrd_create(&t1, f, NULL);
    thrd_create(&t2, g, NULL);
    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
    printf("%d\n", global);
}

```

This program has a write-write data race. Running it through the interpreter results in a single answer, with no errors detected:

```

$ kcc simpleRace.c
$ ./a.out
2

```

However, doing a search with the rules for data race detection given in Section 5.3.1 yields three possible scenarios:⁸

⁸ We use `THREADSEARCH` because we are only interested in nondeterminism stemming from different threads, not nondeterminism inherent in the language. This cuts down the state space significantly. If full nondeterministic search is required, `SEARCH` can be used instead.

```

$ THREADSEARCH=1 GRAPH=1 ./a.out
3 solutions found
-----
Solution 1
Program got stuck
Error: 00049
Description: Have a write-write datarace.
-----
Solution 2
Program completed successfully
Return value: 0
Output:
2
-----
Solution 3
Program completed successfully
Return value: 0
Output:
1

```

Taking a look at the search graph generated by our tool, as shown in Figure 5.1, we can see the sequences of rules that led to the different outcomes: Two paths lead to program termination, but one path results in a data race being detected. Despite the successful paths, the program itself is undefined because of the data race.

To look at a more complicated example, we take a concurrent program from Șerbănuță [146], a parallel implementation of quicksort. We give an updated version using C11 concurrency primitives in Figure 5.2. When called with the appropriate arguments, this program will sort a list beginning at `((qsort_arg*)arg)->b` and ending at `((qsort_arg*)arg)->e`. However, without the commented-out calls to `thrd_join` at the bottom of the function, the call might return before the subthreads are finished. This leads to a race condition if the function that called `quickSort` tries to use the data before it is finished sorting. Given the following `main` function:

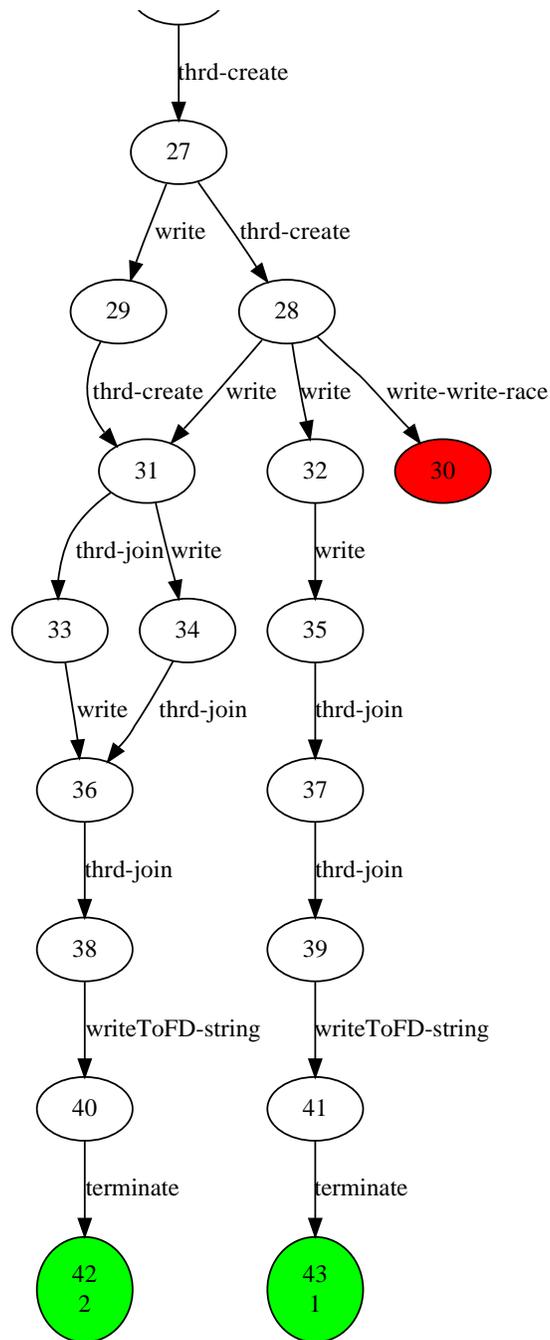


Figure 5.1: Search graph for simpleRace.c

```

typedef struct {
    int *b;
    int *e;
} qsort_arg;

void quickSort(void* arg) {
    int* b = ((qsort_arg*)arg)->b;
    int* e = ((qsort_arg*)arg)->e;

    int t;
    if (! (e <= b + 1)) {
        int p = *b; int *l = b+1; int *r = e;
        while (l + 1 <= r) {
            if (*l <= p) {
                l = l + 1;
            } else {
                r = r - 1;
                t = *l; *l = *r; *r = t;
            }
        }
        l = l - 1;
        t = *l; *l = *b; *b = t;

        qsort_arg* arg1 = malloc(sizeof(qsort_arg));
        arg1->b = b; arg1->e = l;
        qsort_arg* arg2 = malloc(sizeof(qsort_arg));
        arg2->b = r; arg2->e = e;

        thrd_t t1, t2;
        thrd_create(&t1, quickSort, arg1);
        thrd_create(&t2, quickSort, arg2);
        /* thrd_join(t1, NULL);
           thrd_join(t2, NULL); */
    }
}

```

Figure 5.2: Parallel implementation of quicksort

```

int main(void){
    int myArray[] = {77, 19, 12, 15};
    int arrayLen = sizeof(myArray) / sizeof(myArray[0]);
    int* array = malloc(sizeof(int) * arrayLen);
    for (int i = 0; i < arrayLen; i++){
        array[i] = myArray[i];
    }

    quickSort(&((qsort_arg){&array[0], &array[arrayLen]}));

    printf("%d", array[0]);
}

```

we see that executing the program fails to find a bug (although it does return an unsorted first element):

```

$ kcc quicksort.c
$ ./a.out
15

```

Running the same program through search, and the tool detects the data race with similar output as the `simpleRace.c` example above. Uncommenting out the `thrd_join` calls, and the search runs with only a single outcome found.

5.5 Evaluation

In this section we evaluate the semantics-based approach against special-purpose analysis tools. To do so, we explain our testing methodology, which includes a third-party suite of undefined tests as well as a suite of tests we developed.

5.5.1 Third Party Evaluation

In order to evaluate our analysis tool, we first looked for a suite of undefined programs. Although we were unable to find any test suite focusing on undefined behaviors, we did find test suites that included a few key behaviors. Below we briefly mention work we encountered that may evolve into or develop

a complete suite in the future, as well as one suite that we use as a partial undefinedness benchmark.

Related Test Suites

There is a proposed ISO technical specification for program analyzers for C [82], suggesting programmatically enforceable rules for writing secure C code. Many of these rules involve avoiding undefined behavior; however, the specification only focuses on statically enforceable rules. The above classification is similar to MISRA-C [110], whose goal was to create a “restricted subset” of C to help those using C meet safety requirements. MISRA released a “C Exemplar Suite”, containing code both conforming and non-conforming code for the majority of the MISRA C rules. However, these tests contain many undefined behaviors mixed into a single file, and no way to run the comparable defined code without running the undefined code. Furthermore, most of the MISRA tests test only statically detectable undefined behavior. The CERT C Secure Coding Standard [144] and MITRE’s “common weakness enumeration” (CWE) classification system [111] are other similar projects, identifying many causes of program error and cataloguing their severity and other properties. The projects mentioned above include many undefined behaviors—for example, the undefinedness of signed overflow [81, §6.5:5] corresponds to CERT’s INT32-C and to MITRE CWE-190.

Juliet Test Suite

NIST has released a suite of tests for security called the Juliet Test Suite for C/C++ [116], which is based on MITRE’s CWE classification system. It contains over 45,000 tests, each of which triggers one of the 116 different CWEs supported by the suite. Most of the tests (~70%) are C and not C++. However, again the Juliet tests focus on statically detectable violations, and not all of the CWEs are actually undefined—many are simply insecure or unsafe programming practices.

Because the Juliet tests include a single undefined behavior per file and come with positive tests corresponding to the negative tests, we decided to extract an undefinedness benchmark from them. To use the Juliet tests as a test suite for undefinedness, we had to identify which tests were actually undefined. This was largely a manual process that involved understanding the meaning

of each CWE. It was necessary due to the large number of defined-but-bad-practice tests that the suite contains. Interestingly, the suite contained some tests whose supposedly defined portions were actually undefined. Using our analysis tool, we were able to identify six distinct problems with these tests and have submitted the list to NIST. We have not heard back from them yet. We also wrote a small script automating the process of extracting and in some cases fixing the tests, available at <http://code.google.com/p/c-semantics/source/browse/trunk/tests/juliet/clean.sh>.

This extraction gave us 4113 tests, with about 96 SLOC per test (179 SLOC with the helper-library linked in). The tests can be divided into six classes of undefined behavior: use of an invalid pointer (buffer overflow, returning stack address, etc.), division by zero, bad argument to `free()` (stack pointer, pointer not at start of allocated space, etc.), uninitialized memory, bad function call (incorrect number or type of arguments), or integer overflow.

We then ran these tests using a number of analysis tools, including our own semantics-based tool `kcc`. These tools include Valgrind [115],⁹ CheckPointer [99],¹⁰ and the Value Analysis plugin for Frama-C [23].¹¹ Although the Juliet tests are designed to exercise static analysis tools, all of the tools we tested can be considered dynamic analysis tools.¹² The results of this benchmark can be seen in Figure 5.3. Valgrind, and Value Analysis each took, on average, 0.5 s to run the tests, `kcc` took 23 s, and CheckPointer took 80 s. CheckPointer has a large, fixed startup time as it is mainly used to check large software projects, not 100 line programs.

Based on initial results of these tests, we improved our tool to catch precisely those behaviors we were missing. We also contacted the authors of the Value Analysis plugin with their initial results, and they were able to patch their tool within a few days to do the same thing. Because not all tools had this opportunity, the test results should not be taken as any kind of authoritative ranking, but instead suggest some ideas. First, no tool was able to catch behaviors accurately unless they specifically focused on those behaviors. This reaffirms the idea that undefinedness checking does not simply come for free (e.g., by simply leaving out cases), but needs to be studied and understood

⁹v. 3.5.0, <http://valgrind.org/>

¹⁰v. 1.1.5, <http://www.semdesigns.com/Products/MemorySafety/>

¹¹v. Nitrogen-dev, <http://frama-c.com/value.html>

¹²Frama-C's value analysis can be used in "C interpreter" mode [35, §2.1].

Undefined Behavior	No. Tests	Tools (% passed)			
		Valgrind	CheckPointer	V. Analysis	kcc
Use of invalid pointer	3193	70.9	89.1	100.0	100.0
Division by zero	77	0.0	0.0	100.0	100.0
Bad argument to <code>free()</code>	334	100.0	99.7	100.0	100.0
Uninitialized memory	422	100.0	29.3	100.0	100.0
Bad function call	46	100.0	100.0	100.0	100.0
Integer overflow	41	0.0	0.0	100.0	100.0

Figure 5.3: Comparison of analysis tools on Juliet Test Suite

specifically. For example, Valgrind does not try to detect division by zero or integer overflow, and CheckPointer was not designed to detect division by zero, uninitialized memory, or integer overflow. This shows up very clearly in the test results. Second, tools were able to improve performance simply by looking at concrete failing tests and adapting their techniques. As an example, on its initial run on the Juliet tests, `kcc` only caught about 93%.

These ideas mean it is critical that undefinedness benchmarks continue to be developed and used to refine analysis tools. Both we and the Value Analysis team found the Juliet tests useful in improving our tools; in many cases, they gave concrete examples of missing cases that were otherwise hard to identify. The identification, together with the techniques described in Section 5.3, enabled us to adapt our tools to catch every behavior in the suite.

5.5.2 Undefinedness Test Suite

Because we were unable to find an ideal test suite for evaluating detection of undefined behaviors, we began development of our own. This involved first trying to understand the behaviors, and then constructing test cases corresponding to each behavior.

Our Classifications

To help develop our test suite, we first tried to understand the undefined behaviors listed in the standard. Part of this involves classifying the behaviors into categories depending on difficulty. For example, the standard says: “The (nonexistent) value of a `void` expression (an expression that has type `void`) shall not be used in any way, and implicit or explicit conversions (except to `void`) shall not be applied to such an expression” [81, §6.3.2.2:1]. Depending on how one interprets the word “use”, this could be a static or dynamic restriction. If static, the code:

```
if (0) { (int)(void)5; }
```

is undefined according to §6.3.2.2:1; if dynamic, it is defined since the problematic code can never be reached. The intention behind the standard¹³ appears to be that, in general, situations are made statically undefined if it is not easy to generate code for them. Only when code can be generated, then

¹³Private correspondence with committee member.

the situation can be undefined dynamically. In the above example, it is hard to imagine code being generated for `(int)(void)5`, so we can conclude this is meant to be statically undefined. When there was any confusion as to the static/dynamic nature of any of the behaviors, we use the above assumption.

We found that the majority of the categories of undefined behavior in C are dynamic in nature. Out of 221 undefined behaviors, 92 are statically detectable and 129 are only dynamically detectable. Because the argument for the undecidability of detecting undefinedness (Section 5.2.6) does not depend on the particular dynamic behavior, detecting any dynamic behavior is equally hard. This does not apply to the static behaviors, as they are undefined for static reasons and are not subject to particular control flows.

Our Test Suite

An ideal test suite for undefined behaviors involves individual tests for each of the 221 undefined behaviors. Some behaviors require multiple tests, e.g., “If the specification of a function type includes any type qualifiers, the behavior is undefined.” [81, §6.7.3:9] requires at least one test for each qualifier. Ideally the tests would also include control-flow, data-flow, and execution-flow variations in order to make static analysis more difficult.

As we discussed in Section 5.2.4, dynamic undefined behavior on a reachable path (or any statically undefined behavior) causes the entire program to become undefined. This means that each test in the test suite needs to be a separate program, otherwise one undefined behavior may interact with another undefined behavior. In addition, each test should come with a corresponding defined test. This “control” test makes it possible to identify false-positives in addition to false-negatives. Without such tests, a tool could simply say all programs were undefined and receive full marks.

Our suite currently includes 178 tests covering 70 of the undefined behaviors. We hope it will serve as a starting point for the development of a larger, more comprehensive test. Our suite focuses almost entirely on the non-library behaviors, and specifically on the dynamic behaviors therein. It includes at least one test for each of the 42 dynamically undefined behaviors relating to the non-library part of the language that are not also implementation-specific. We have made our test suite and categorization available for download at <http://code.google.com/p/c-semantics/downloads/>.

These tests are much broader than the Juliet tests, covering 70 undefined behaviors as opposed to the 6 covered by the Juliet tests. However, each behavior is tested shallowly, with only 2 tests per behavior on average. Some of the dynamic behaviors it tests that the Juliet suite does not include:

- If the program attempts to modify [a character string literal], the behavior is undefined. [81, §6.4.5:7]
- An object shall have its stored value accessed only by an lvalue expression that has [an allowed type]. [81, §6.5:7]
- When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object. [81, §6.5.6:9]
- 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. [81, §6.5:2]

There are many other such behaviors tested, and all are equally bad from the C standard’s perspective. They can all cause a compiler to generate unexpected code or cause a running program to behave in an unexpected way.

We compared the same tools as before using our own custom made tests. The results can be seen in Figure 5.4. It is clear that the tools focusing on a few common undefined behaviors (Valgrind and CheckPointer) only detect a small percentage of behaviors. Both Value Analysis and `kcc`, which were designed to catch a large number of undefined behaviors, were able to catch a much larger number of dynamic behaviors, and in the case of `kcc`, many of the static behaviors as well.

5.6 Conclusion

In this chapter we investigated undefined behaviors in C and how one can capture these behaviors semantically. We discussed three techniques for formally describing undefined behaviors. We also used these techniques in a semantics-based analysis tool, which we tested against other popular analysis

Tools	Static (% Passed)	Dynamic (% Passed)
Valgrind	0.0	2.3
V. Analysis	1.6	45.3
CheckPtr.	2.4	13.1
kcc	44.8	64.0

Figure 5.4: Comparison of analysis tools against our tests. These averages are across undefined behaviors, and no behavior is weighted more than another.

tools. We compared the tools on a test suite of our own devising, which we are making publicly available, as well as on another publicly available test suite.

We hope that this work will bring more attention to the problem of undefined behavior in program verification. Undefined programs may behave in *any* way and undefinedness is (in general) undecidable to detect; this means that undefined programs are a serious problem that needs to be addressed by analysis tools. Whether this is through semantic means or some other mechanism, tools to verify the absence of undefined behavior are needed on the road to fully verified software.

Chapter 6

Conclusion

In the previous chapters, we explained how a complete formal semantics for C can be defined in the \mathbb{K} framework, yielding tools for execution and analysis. We now explain some of the limitations of our work, as well as obvious extensions. We finish with some forward-looking words on the state of the formal semantics discipline.

6.1 Limitations

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

There are two main ways in which semantics can be incomplete—under-definedness and over-definedness. Typically when one thinks of incompleteness, one thinks of failure to give meaning to correct programs. However, because we want to be able to identify incorrect or unportable programs, the semantics must be balanced appropriately between defining too much or too little. It is equally important not to give semantics to programs that should be undefined.

In the first case, we are not missing any features—we have given semantics to every feature required of a freestanding implementation of C. With this said, our semantics is not perfect. For example, we still are not passing 100% of our test cases (see Section 4.3). Also, our semantics of floating point numbers is particularly weak. During execution or analysis, we simply rely on an IEEE-754 implementation of floating point arithmetic provided to us by our definitional framework (\mathbb{K}). This is fine for interpretation and explicit state model checking, but not for deductive reasoning.

In the second case, although our semantics can catch many bad behaviors other tools cannot (e.g., we have not found any other tool that catches the undefined programs in Sections 5.3.3 or 4.4.2), there is still room for improvement. For one, our semantics aligns all types to one-byte boundaries. This

means we cannot catch undefined behavior related to alignment restrictions. Note that others have worked on formalizing alignment requirements [117], but it has never been incorporated into a full semantics for C. We also do not handle the type qualifiers `volatile` or `restrict`; we simply ignore them. This is safe to do when interpreting correct programs, but it means we are not detecting problems related to those features in incorrect programs. It also means that we are missing possible behaviors when searching programs that use `volatile`.

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 [139] and semantics level proofs [50]. 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. This is already being done for a subset of the C language [135], and we intend to apply those techniques to actual C in the future.

We still do not cover all of the standard library headers. So far, we have added library functions by need in order to run example programs, which is why we have semantics for library functions like `malloc()`, `longjmp()`, parts of `printf()`, variadic functions, and over 30 others. We intend on covering more libraries in the future, but for now, one could supplement what we provide by using implementations of libraries written in C.

In our current semantics, only some of the implementation-defined behaviors are available—the most common ones. By making the semantics parametric, we hope others can add or change implementation-defined rules to suit their needs.

Finally, we should mention the speed of our system. While it is not nearly as fast as C compiled natively, it is usable. Of the GCC torture test programs described listed in Section 4.3, our semantics ran over 93% of these programs in under 10 seconds (each). An additional 4% completed in 2 minutes, 2% in 5 hours, and 1% further in under 3 days. In comparison, it takes GCC about 0.05s for each test. The reader should keep in mind that this is an interpreter obtained for free from a formal semantics. In addition, the search and model checking tools suffer the same state explosion problems inherent in all explicit-state model checking.

6.2 Future Work

In this section, we describe some of the next steps one could take developing our semantics and the tools derived from the semantics.

6.2.1 Semantics

While the positive semantics for C99 features is complete, there are still C11 features we need to add. These include `_Alignas`, static-assertions, anonymous `structs` and `unions`, type-generic expressions, `_Atomic`, `stdatomic.h`, and many functions from `threads.h`. There are also some remaining bugs to be fixed in the C99 semantics, as uncovered by the GCC torture tests in Section 4.3.

There are also many negative-semantics issues that can be addressed, as uncovered by the evaluation in Section 5.5. In particular, we would like to handle programs that are invalid due to strict aliasing [81, §6.5:7], which we do not currently catch in all cases. Strict aliasing is often confusing and unexpected to those who are not aware of it, causing problems in otherwise correct code [1].

A complete list of the missing bugs and features is available at <http://code.google.com/p/c-semantics/issues/list>.

There are many extensions to C that could be formalized as well. These include GCC extensions [59, §6], Cilk concurrency primitives [14], and Apple blocks (closures for C) [5], among many others. One such extension has already been specified on top of our semantics by a third party: Chris Hathhorn added CUDA extensions to our semantics [71] for parallel computing using graphics processing units (GPUs).

6.2.2 Tools

In terms of work related to tools, there are two major tasks that need to be addressed. The first is applying matching logic [136, 137, 139] to the full C semantics. While there is a matching logic tool available [135] for a subset of C, it has not yet been extended to the complete semantics. Doing this involves first developing a generic tool, so that given a mechanism for annotations and a \mathbb{K} definition, it supports all the connective tissue needed to connect the proof exploration mechanisms with the rules of the language. Second,

specific abstraction techniques need to be developed for features of C that have not yet been covered by our previous tools, such as arrays.

The other major task involves a way to abstract the state space for state-space search or model checking. In particular, a way to identify states that only disagree via renaming of addresses would be extremely helpful in cutting down the state space of programs. For example, currently, code with local variables causes a counter for memory locations to be incremented, changing the location of other variables in the future. While these states might all be equivalent under a renaming of locations, they are not identical, and so the current tool treats these as different states, contributing to the state-space explosion problem.

6.3 Conclusion

It is a shame that, despite the best efforts of over 40 years of research in formal programming languages, most language designers still consider the difficulties of defining formal semantics to outweigh the benefits. Formal semantics and practicality are not typically considered together. When C was being standardized, the standards committee explored using formal semantics, but in the end 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” [80, §6]. This is a common sentiment in the programming language community. Indeed, startlingly few “real” languages have ever been completely formalized, and even fewer were designed with formal specification in mind.

Based on our experience with our semantics, the development of a formal semantics for C could have taken place alongside the development of the standard. Within roughly 6 person-months, we had a working version of our semantics that covered more of the standard than any previous semantics. The version presented in this dissertation is the result of 18 person-months of work. To put this in perspective, one member of the standards committee estimated that it took roughly 62 person-*years* to produce the C99 standard [83, p. 6]. We are *not* claiming that we have done the same job in a fraction of the time; obviously writing a semantics based on the standard is quite different than writing the standard itself. We are simply saying that the effort it takes to

develop a rewriting-based semantics is quite small compared to the effort it took to develop the standard.

The reluctance of the language community towards formal methods has not been without reason—it is not always clear that having a formal semantics earns the designer anything tangible for her effort. Commonly mentioned benefits like improving the understanding of the language or providing a model in which sound arguments about the language can be made are relatively intangible; to be accepted by the general language community, semantics needs to be shown to have concrete value beyond that of prose.

The time has come to start building analysis tools directly on formal models. Instead of building analysis tools for different languages and different versions of each language, the analysis infrastructure surrounding the semantics could be maintained independently so that one could derive tools for multiple languages simply by swapping out the semantic rules. Formal analysis tools would be safer than traditional tools based on informal models of the target languages. In addition, as the language changes and the specifications change, the semantics can be changed in a single place and the tools regenerated. We offer our work as one small step in this direction; we are not alone, and there are other tools including pluggable analysis architectures like Frama-C [34] and formal tools like CompCert [13] that share part of this vision.

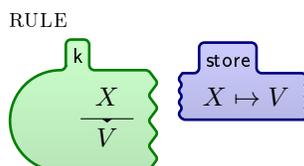
Appendix A

Entire Annotated Semantics

In this appendix, we present the full dynamic semantics of C as expressed in the \mathbb{K} Framework. We use a slightly different notation here than in the rest of this dissertation, as it makes the rules easier to read at a glance. This notation was not used in previous chapters as it often takes up more space. The only thing different about this notation is the way cells are represented—now, cells completely graphically surround their contents (instead of using XML-like brackets), and elision is represented using torn cell edges instead of \dots . For example, this rule:

$$\frac{\langle X \dots \rangle_k}{V} \quad \langle \dots X \mapsto V \dots \rangle_{\text{store}}$$

is equivalent to this rule:



In this appendix, comments to the formal semantics are set inside gray boxes. These boxes often contain excerpts from the most recent draft of the C11 standard, and begin with the draft number in parentheses, followed by the relevant section and paragraph (e.g., (n1570) §6.3.2.1¶2). All such comments are copyright ISO/IEC and are being used here solely for educational purposes as a matter of comparison.

A.1 Syntax

This section presents the \mathbb{K} syntax of C. While much of the syntax is given as abstract, prefix productions, the majority of expressions are given as mixfix productions. This allows expression constructs to look as close as possible to the original syntax without making the parsing of semantic rules difficult for \mathbb{K} .

It is important to notice the strictness annotations on the rules, as described in Section 3.2. These annotations describe the allowed order of evaluation for C constructs. Evaluation contexts are used when strictness is not expressive enough. Attributes like `ndheat` (and later `ndlocal`, synonyms for superheating and supercooling, respectively) are used to help carve out a search space. Please see Şerbănuţă et al. [150] for more details on superheating and supercooling.

As a matter of convention, operators starting with an uppercase letter are syntactic—they represent constructs coming directly from the language. Operators starting with a lowercase letter are semantic—they are constructs used to represent partially evaluated computations, helper operators, or functions.

MODULE COMMON-C-SYNTAX

SYNTAX $TypeSpecifier ::= AlignasExpression(K)$
 | $AlignasType(K, K)$

SYNTAX $DeclType ::= ArrayType(K, K, K)$ [strict(1)]

SYNTAX $K ::= AnonymousName$

SYNTAX $Id ::= \#NoName$
 | $\#NoName(Nat)$ [klabel(#NoName)]

SYNTAX $Expression ::= - K$ [type-strict]
 | $+ K$ [type-strict]
 | $! K$ [type-strict]
 | $* K$ [type-strict]
 | $\& K$ [strict type-strict]
 | $++ K$ [type-strict]
 | $-- K$ [type-strict]
 | $AlignofExpression(K)$
 | $AlignofType(K, K)$
 | $ArrayIndex(K, K)$
 | $Arrow(K, Id)$
 | $Assign(K, K)$
 | $AssignBitwiseAnd(K, K)$
 | $AssignBitwiseOr(K, K)$
 | $AssignBitwiseXor(K, K)$
 | $AssignDivide(K, K)$
 | $AssignLeftShift(K, K)$
 | $AssignMinus(K, K)$
 | $AssignModulo(K, K)$
 | $AssignMultiply(K, K)$
 | $AssignPlus(K, K)$
 | $AssignRightShift(K, K)$

SYNTAX $C ::= AttributeWrapper(K, K)$

SYNTAX $TypeSpecifier ::= Atomic$

SYNTAX $Storage ::= \text{Auto}$

SYNTAX $FieldName ::= \text{BitFieldName}(K, K)$

SYNTAX $KResult ::= \text{AtIndexInit}(K, K)$
 $\quad \quad \quad | \text{AtIndexRangeInit}(K, K)$

SYNTAX $C ::= \text{Attribute}(String, K)$

SYNTAX $Expression ::= \text{BitwiseAnd}(K, K)$
 $\quad \quad \quad | \text{BitwiseNot}(K)$
 $\quad \quad \quad | \text{BitwiseOr}(K, K)$
 $\quad \quad \quad | \text{BitwiseXor}(K, K)$

SYNTAX $CabsLoc ::= \text{CabsLoc}(String, Int, Int, Int)$

SYNTAX $TypeSpecifier ::= \text{Char}$
 $\quad \quad \quad | \text{Bool}$
 $\quad \quad \quad | \text{Complex}$

SYNTAX $CVSpecifier ::= \text{Const}$

SYNTAX $PureDefinition ::= \text{DeclarationDefinition}(K)$

SYNTAX $Definition ::= \text{DefinitionLoc}(K, K)$
 $\quad \quad \quad | \text{DefinitionLocRange}(K, K, K)$

SYNTAX $Block ::= \text{Block}(Nat, K, K)$

SYNTAX $PureStatement ::= K;$
 $\quad \quad \quad | \text{BlockStatement}(K)$
 $\quad \quad \quad | \text{Break}$
 $\quad \quad \quad | \text{Continue}$
 $\quad \quad \quad | \text{Case}(Nat, Nat, K, K)$
 $\quad \quad \quad | \text{CaseRange}(K, K, K)$
 $\quad \quad \quad | \text{Default}(Nat, K)$
 $\quad \quad \quad | \text{CompGoto}(K)$

SYNTAX $Expression ::= \text{Cast}(K, K, K) [\text{strict}(1)]$

- | CompoundLiteral(K, K, K, K) [strict(2)]
- | Call(K, K) [type-strict(1)]
- | Comma(K)
- | Constant(K)

SYNTAX *IntConstant* ::= DecimalConstant(K)
| DecimalFloatConstant(*String*, *Int*, *Float*)

SYNTAX *Constant* ::= CharLiteral(*Int*)

SYNTAX *InitExpression* ::= CompoundInit(K) [hybrid strict]

SYNTAX K ::= CodeLoc(K, K)

SYNTAX *Expression* ::= Conditional(K, K, K)
| Dereference(K)
| Divide(K, K)

SYNTAX *PureStatement* ::= DoWhile(K, K)

SYNTAX *Expression* ::= Dot(K, Id)

SYNTAX *TypeSpecifier* ::= Double
| EnumRef(*Id*)
| EnumDef(*Id*, K)

SYNTAX *PureEnumItem* ::= EnumItem(*Id*)
| EnumItemInit(*Id*, K)

SYNTAX *Expression* ::= Equality(K, K)

SYNTAX *TypeSpecifier* ::= Float

SYNTAX *Storage* ::= Extern

SYNTAX *DeclType* ::= FunctionType(K) [strict]

SYNTAX *FieldGroup* ::= FieldGroup(K, K) [strict(1)]

- | LTL-And(K, K)
- | LTL-Or(K, K)
- | LTL-Not(K)
- | LTL-Next(K)
- | LTL-Always(K)
- | LTL-Eventually(K)
- | LTL-Until(K, K)
- | LTL-Release(K, K)
- | LTL-Implies(K, K)
- | LTL-Equiv(K, K)
- | LTL-WeakUntil(K, K)

SYNTAX *PureStatement* ::= IfThenElse(K, K, K)
 | Label(Id, K)

SYNTAX *IntConstant* ::= HexConstant(K)
 | HexFloatConstant($String, Int, Float$)

SYNTAX *Constant* ::= L(K)
 | LL(K)

SYNTAX *InitFragment* ::= InitFragment(K, K)

SYNTAX *KResult* ::= InFieldInit(Id, K)

SYNTAX *Expression* ::= LeftShift(K, K)
 | LessThan(K, K)
 | LessThanOrEqual(K, K)

SYNTAX *C* ::= List($List\{K\}$)

SYNTAX *PureDefinition* ::= Linkage($String, K$)

SYNTAX *Expression* ::= LogicalAnd(K, K)
 | LogicalNot(K)
 | LogicalOr(K, K)

SYNTAX *TypeSpecifier* ::= Long

SYNTAX $Expression ::= \text{Minus}(K, K)$
 | $\text{Modulo}(K, K)$
 | $\text{Multiply}(K, K)$

SYNTAX $TypeSpecifier ::= \text{Named}(Id)$

SYNTAX $Storage ::= \text{NoStorage}$

SYNTAX $SpecifierElem ::= \text{Noreturn}$

SYNTAX $NameGroup ::= \text{NameGroup}(K, K)$ [strict(1)]

SYNTAX $Name ::= \text{Name}(K, K)$

SYNTAX $PureStatement ::= \text{Nop}$

SYNTAX $Constant ::= \text{NoSuffix}(K)$

SYNTAX $KResult ::= \text{NoInit}$
 | NextInit

SYNTAX $Expression ::= \text{Negative}(K)$
 | $\text{NotEquality}(K, K)$

SYNTAX $K ::= \text{NotVariadic}$

SYNTAX $PureDefinition ::= \text{OnlyTypedef}(K)$

SYNTAX $Expression ::= \text{OffsetOf}(K, K, K)$ [strict(1)]
 | NothingExpression

SYNTAX $IntConstant ::= \text{OctalConstant}(K)$

SYNTAX $Expression ::= \text{Plus}(K, K)$

SYNTAX $Storage ::= \text{Register}$

SYNTAX $CVSpecifier ::= \text{Restrict}$

SYNTAX $DeclType ::= \text{PointerType}(K) \text{ [strict]}$
 | $\text{Prototype}(K, K, Bool) \text{ [strict(1)]}$

SYNTAX $PureDefinition ::= \text{Pragma}(K)$

SYNTAX $Program ::= \text{Program}(K)$

SYNTAX $PureStatement ::= \text{Return}(K)$

SYNTAX $Expression ::= \text{Positive}(K)$
 | $\text{Reference}(K)$
 | $\text{PreIncrement}(K)$
 | $\text{PreDecrement}(K)$
 | $\text{PostIncrement}(K)$
 | $\text{PostDecrement}(K)$
 | $\text{RightShift}(K, K)$

SYNTAX $K ::= \text{reval}(K) \text{ [semantic strict]}$
 | $\text{peval}(K) \text{ [semantic strict]}$

SYNTAX $C ::= Id$

SYNTAX $K ::= \text{StmtCons}(K, K)$

this production ensures that a `TypeResult` sort is created, together with an `'isTypeResult` predicate

SYNTAX $TypeResult ::= \text{dummyTypeProduction}$

SYNTAX $KResult ::= \text{SpecifierElem}$

SYNTAX $C ::= CabsLoc$
 | $TypeSpecifier$
 | $Storage$
 | $CVSpecifier$
 | $SpecifierElem$
 | $Specifier$
 | $DeclType$
 | $NameGroup$

| *FieldGroup*
 | *InitNameGroup*
 | *Name*
 | *InitName*
 | *SingleName*
 | *Definition*
 | *Block*
 | *Statement*
 | *PureStatement*
 | *PureEnumItem*
 | *ForClause*
 | *Expression*
 | *Constant*
 | *InitExpression*
 | *Program*
 | *TranslationUnit*
 | *IntConstant*
 | *InitFragment*
 | *FieldName*
 | *PureDefinition*

SYNTAX *TypeSpecifier* ::= Void
 | Short
 | Signed
 | Unsigned

SYNTAX *Float* ::= inf

SYNTAX *TypeSpecifier* ::= StructRef(*Id*)
 | StructDef(*Id*, *K*)

CONTEXT: StructDef(—, List(—, □, —))

SYNTAX *TypeSpecifier* ::= UnionRef(*Id*)
 | UnionDef(*Id*, *K*)

CONTEXT: UnionDef(—, List(—, □, —))

SYNTAX *TypeSpecifier* ::= TypeofExpression(*K*)

| TypeofType(K, K)
 | TAtomic(K, K)

SYNTAX *Storage* ::= Static
 | ThreadLocal

SYNTAX *CVSpecifier* ::= Volatile

SYNTAX *SpecifierElem* ::= SpecTypedef
 | *CVSpecifier*
 | *Storage*
 | *TypeSpecifier*
 | SpecPattern(*Id*)

SYNTAX *Specifier* ::= Specifier(K)

CONTEXT: Specifier(List($-$, \square , $-$))

CONTEXT: ArrayType($-$, $\frac{\square}{\text{reval}(\square)}$, $-$)[ndheat]

CONTEXT: Prototype($-$, List($-$, \square , $-$), $-$)

SYNTAX K ::= Variadic

CONTEXT: NameGroup($-$, List($-$, \square , $-$))

MACRO

AnonymousName = #NoName

CONTEXT: InitName($-$, $\frac{\square}{\text{reval}(\square)}$)[ndheat]

SYNTAX *SingleName* ::= SingleName(K, K) [strict(1)]

SYNTAX *PureDefinition* ::= Typedef(K)
 | Transformer(K, K)

SYNTAX *TranslationUnit* ::= TranslationUnit(*String*, K, K, \textit{String})

CONTEXT: $\frac{\square}{\text{reval}(\square)}$;

SYNTAX *PureStatement* ::= Sequence(*K*, *K*)

CONTEXT: IfThenElse($\frac{\square}{\text{reval}(\square)}$, —, —)

SYNTAX *PureStatement* ::= While(*K*, *K*)

CONTEXT: Return($\frac{\square}{\text{reval}(\square)}$)

SYNTAX *PureStatement* ::= Switch(*K*, *K*, *K*)

CONTEXT: Switch(—, $\frac{\square}{\text{reval}(\square)}$, —)

SYNTAX *Statement* ::= StatementLoc(*K*, *K*)

CONTEXT: - $\frac{\square}{\text{reval}(\square)}$

CONTEXT: + $\frac{\square}{\text{reval}(\square)}$

CONTEXT: ! $\frac{\square}{\text{reval}(\square)}$

SYNTAX *Expression* ::= $\sim K$ [type-strict]

CONTEXT: $\sim \frac{\square}{\text{reval}(\square)}$

CONTEXT: * $\frac{\square}{\text{reval}(\square)}$

SYNTAX *Expression* ::= *K* ++ [type-strict]

CONTEXT: $\frac{\square}{\text{peval}(\square)} ++$

SYNTAX $Expression ::= K -- [\text{type-strict}]$

CONTEXT: $\frac{\square}{\text{peval}(\square)} --$

SYNTAX $Expression ::= K * K [\text{type-strict}]$
 $\quad \quad \quad | K / K [\text{type-strict}]$
 $\quad \quad \quad | K \% K [\text{type-strict}]$

CONTEXT: $\frac{\square}{\text{reval}(\square)} * \text{---} [\text{ndheat}]$

CONTEXT: $\text{---} * \frac{\square}{\text{reval}(\square)} [\text{ndheat}]$

CONTEXT: $\frac{\square}{\text{reval}(\square)} / \text{---} [\text{ndheat}]$

CONTEXT: $\text{---} / \frac{\square}{\text{reval}(\square)} [\text{ndheat}]$

CONTEXT: $\frac{\square}{\text{reval}(\square)} \% \text{---} [\text{ndheat}]$

CONTEXT: $\text{---} \% \frac{\square}{\text{reval}(\square)} [\text{ndheat}]$

SYNTAX $Expression ::= K + K [\text{type-strict}]$
 $\quad \quad \quad | K - K [\text{type-strict}]$

CONTEXT: $\frac{\square}{\text{reval}(\square)} + \text{---} [\text{ndheat}]$

CONTEXT: $\text{---} + \frac{\square}{\text{reval}(\square)} [\text{ndheat}]$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} \text{---} \text{[ndheat]}$

CONTEXT: $\text{---} \frac{\Box}{\text{reval}(\Box)} \text{[ndheat]}$

SYNTAX $Expression ::= K \ll K \text{ [type-strict(1)]}$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} \ll \text{---} \text{[ndheat]}$

CONTEXT: $\text{---} \ll \frac{\Box}{\text{reval}(\Box)} \text{[ndheat]}$

SYNTAX $Expression ::= K \gg K \text{ [type-strict(1)]}$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} \gg \text{---} \text{[ndheat]}$

CONTEXT: $\text{---} \gg \frac{\Box}{\text{reval}(\Box)} \text{[ndheat]}$

SYNTAX $Expression ::= K < K \text{ [type-strict]}$
 $\quad \quad \quad | K <= K \text{ [type-strict]}$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} < \text{---} \text{[ndheat]}$

CONTEXT: $\text{---} < \frac{\Box}{\text{reval}(\Box)} \text{[ndheat]}$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} <= \text{---} \text{[ndheat]}$

CONTEXT: $\text{---} <= \frac{\Box}{\text{reval}(\Box)} \text{[ndheat]}$

SYNTAX $Expression ::= K > K \text{ [type-strict]}$
 $\quad \quad \quad | K >= K \text{ [type-strict]}$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} > \text{---}[\text{ndheat}]$

CONTEXT: $\text{---} > \frac{\Box}{\text{reval}(\Box)} [\text{ndheat}]$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} >= \text{---}[\text{ndheat}]$

CONTEXT: $\text{---} >= \frac{\Box}{\text{reval}(\Box)} [\text{ndheat}]$

SYNTAX $Expression ::= K == K [\text{type-strict}]$
 $| K != K [\text{type-strict}]$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} == \text{---}[\text{ndheat}]$

CONTEXT: $\text{---} == \frac{\Box}{\text{reval}(\Box)} [\text{ndheat}]$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} != \text{---}[\text{ndheat}]$

CONTEXT: $\text{---} != \frac{\Box}{\text{reval}(\Box)} [\text{ndheat}]$

SYNTAX $Expression ::= K \& K [\text{type-strict}]$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} \& \text{---}[\text{ndheat}]$

CONTEXT: $\text{---} \& \frac{\Box}{\text{reval}(\Box)} [\text{ndheat}]$

SYNTAX $Expression ::= K \wedge K [\text{type-strict}]$

CONTEXT: $\frac{\Box}{\text{reval}(\Box)} \wedge \text{---}[\text{ndheat}]$

CONTEXT: $\frac{\square}{\text{reval}(\square)} \text{---} \wedge \text{---}$ [ndheat]

SYNTAX $Expression ::= K \mid K$ [type-strict]

CONTEXT: $\frac{\square}{\text{reval}(\square)} \text{---} \mid \text{---}$ [ndheat]

CONTEXT: $\text{---} \mid \frac{\square}{\text{reval}(\square)} \text{---}$ [ndheat]

SYNTAX $Expression ::= K \ \&\& \ K$ [type-strict]

CONTEXT: $\frac{\square}{\text{reval}(\square)} \ \&\& \ \text{---}$ [ndheat]

SYNTAX $Expression ::= K \ \mid\mid \ K$ [type-strict]

CONTEXT: $\frac{\square}{\text{reval}(\square)} \ \mid\mid \ \text{---}$ [ndheat]

SYNTAX $Expression ::= K \ * = \ K$ [type-strict(1)]
| $K \ / = \ K$ [type-strict(1)]
| $K \ \% = \ K$ [type-strict(1)]
| $K \ + = \ K$ [type-strict(1)]
| $K \ - = \ K$ [type-strict(1)]
| $K \ \gg = \ K$ [type-strict(1)]
| $K \ \& = \ K$ [type-strict(1)]
| $K \ \wedge = \ K$ [type-strict(1)]
| $K \ \mid = \ K$ [type-strict(1)]
| $K \ \ll = \ K$ [type-strict(1)]
| $K \ := \ K$ [type-strict(1)]

CONTEXT: $\frac{\square}{\text{peval}(\square)} := \text{---}$ [ndheat]

CONTEXT: $\text{---} := \frac{\square}{\text{reval}(\square)} \text{---}$ [ndheat]

SYNTAX $Expression ::= K ? K : K$ [type-strict(2 3)]

CONTEXT: $\frac{\square}{\text{reval}(\square)} ? - : -$

CONTEXT: $\text{Cast}(-, -, \frac{\square}{\text{reval}(\square)})$

CONTEXT: $\text{Call}(\frac{\square}{\text{reval}(\square)}, -)[\text{ndheat}]$

CONTEXT: $\text{Call}(-, \text{List}(-, \frac{\square}{\text{reval}(\square)}, -))[\text{ndheat}]$

CONTEXT: $\text{Comma}(\text{List}(\frac{\square}{\text{reval}(\square)}, -))[\text{ndheat}]$

SYNTAX $Expression ::= Id$
| $\text{SizeofExpression}(K)$
| $\text{SizeofType}(K, K)$ [strict(1)]
| $K[K]$ [type-strict]
| $K . Id$ [type-strict(1)]

CONTEXT: $\frac{\square}{\text{peval}(\square)} . -$

SYNTAX $Expression ::= K \rightarrow Id$

CONTEXT: $\frac{\square}{\text{reval}(\square)} \rightarrow -$

SYNTAX $Constant ::= U(K)$
| $UL(K)$
| $ULL(K)$
| $WCharLiteral(Int)$
| $StringLiteral(String)$
| $WStringLiteral(List\{K\})$

SYNTAX $InitExpression ::= \text{SingleInit}(K)$ [hybrid strict]

MACRO
DefinitionLoc(K, L) = CodeLoc(K, L)

MACRO
StatementLoc(K, L) = CodeLoc(K, L)

MACRO
DefinitionLocRange($K, _ , L$) = CodeLoc(K, L)

This macro defines an important identity from (n1570) §6.5.3.2 ¶3. As a syntactic macro, it should run on programs before they even start to reduce.

MACRO
& (* K) = K

The below macros simply transform the prefix AST names to the infix/mixfix names we use from now on

MACRO
Conditional(K_1, K_2, K_3) = $K_1 ? K_2 : K_3$

MACRO
ArrayIndex(K_1, K_2) = $K_1[K_2]$

MACRO
Negative(K) = - K

MACRO
Positive(K) = + K

MACRO
LogicalNot(K) = ! K

MACRO
BitwiseNot(K) = ~ K

MACRO
Dereference(K) = * K

MACRO
Reference(K) = & K

MACRO
PreIncrement(K) = ++ K

```
MACRO
PreDecrement( $K$ ) = --  $K$ 

MACRO
PostIncrement( $K$ ) =  $K$  ++

MACRO
PostDecrement( $K$ ) =  $K$  --

MACRO
Multiply( $K_1, K_2$ ) =  $K_1 * K_2$ 

MACRO
Divide( $K_1, K_2$ ) =  $K_1 / K_2$ 

MACRO
Modulo( $K_1, K_2$ ) =  $K_1 \% K_2$ 

MACRO
Plus( $K_1, K_2$ ) =  $K_1 + K_2$ 

MACRO
Minus( $K_1, K_2$ ) =  $K_1 - K_2$ 

MACRO
LeftShift( $K_1, K_2$ ) =  $K_1 \ll K_2$ 

MACRO
RightShift( $K_1, K_2$ ) =  $K_1 \gg K_2$ 

MACRO
LessThan( $K_1, K_2$ ) =  $K_1 < K_2$ 

MACRO
LessThanOrEqual( $K_1, K_2$ ) =  $K_1 <= K_2$ 

MACRO
GreaterThan( $K_1, K_2$ ) =  $K_1 > K_2$ 

MACRO
GreaterThanOrEqual( $K_1, K_2$ ) =  $K_1 >= K_2$ 

MACRO
Equality( $K_1, K_2$ ) =  $K_1 == K_2$ 
```

```
MACRO
NotEquality( $K_1, K_2$ ) =  $K_1 != K_2$ 

MACRO
BitwiseAnd( $K_1, K_2$ ) =  $K_1 \& K_2$ 

MACRO
BitwiseXor( $K_1, K_2$ ) =  $K_1 \wedge K_2$ 

MACRO
BitwiseOr( $K_1, K_2$ ) =  $K_1 | K_2$ 

MACRO
LogicalAnd( $K_1, K_2$ ) =  $K_1 \&\& K_2$ 

MACRO
LogicalOr( $K_1, K_2$ ) =  $K_1 || K_2$ 

MACRO
Assign( $K_1, K_2$ ) =  $K_1 := K_2$ 

MACRO
AssignMultiply( $K_1, K_2$ ) =  $K_1 *= K_2$ 

MACRO
AssignDivide( $K_1, K_2$ ) =  $K_1 /= K_2$ 

MACRO
AssignModulo( $K_1, K_2$ ) =  $K_1 \% = K_2$ 

MACRO
AssignPlus( $K_1, K_2$ ) =  $K_1 += K_2$ 

MACRO
AssignMinus( $K_1, K_2$ ) =  $K_1 -= K_2$ 

MACRO
AssignBitwiseAnd( $K_1, K_2$ ) =  $K_1 \&= K_2$ 

MACRO
AssignBitwiseXor( $K_1, K_2$ ) =  $K_1 \wedge = K_2$ 

MACRO
AssignBitwiseOr( $K_1, K_2$ ) =  $K_1 |= K_2$ 

MACRO
AssignLeftShift( $K_1, K_2$ ) =  $K_1 \gg = K_2$ 
```

```
MACRO
AssignRightShift( $K_1, K_2$ ) =  $K_1 \ll= K_2$ 
```

```
MACRO
Dot( $K, X$ ) =  $K \cdot X$ 
```

```
MACRO
Arrow( $K, X$ ) =  $K \rightarrow X$ 
```

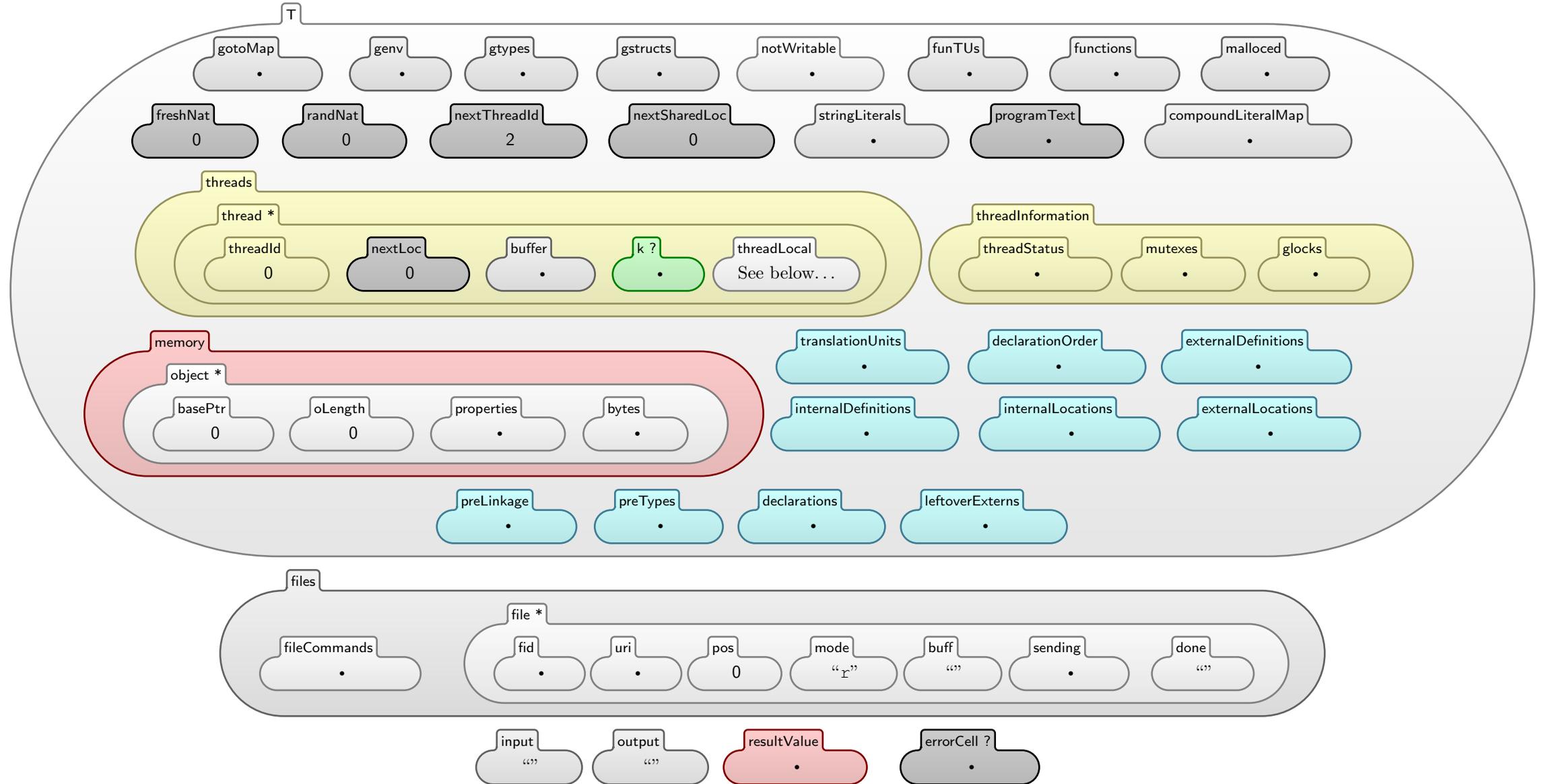
```
END MODULE
```

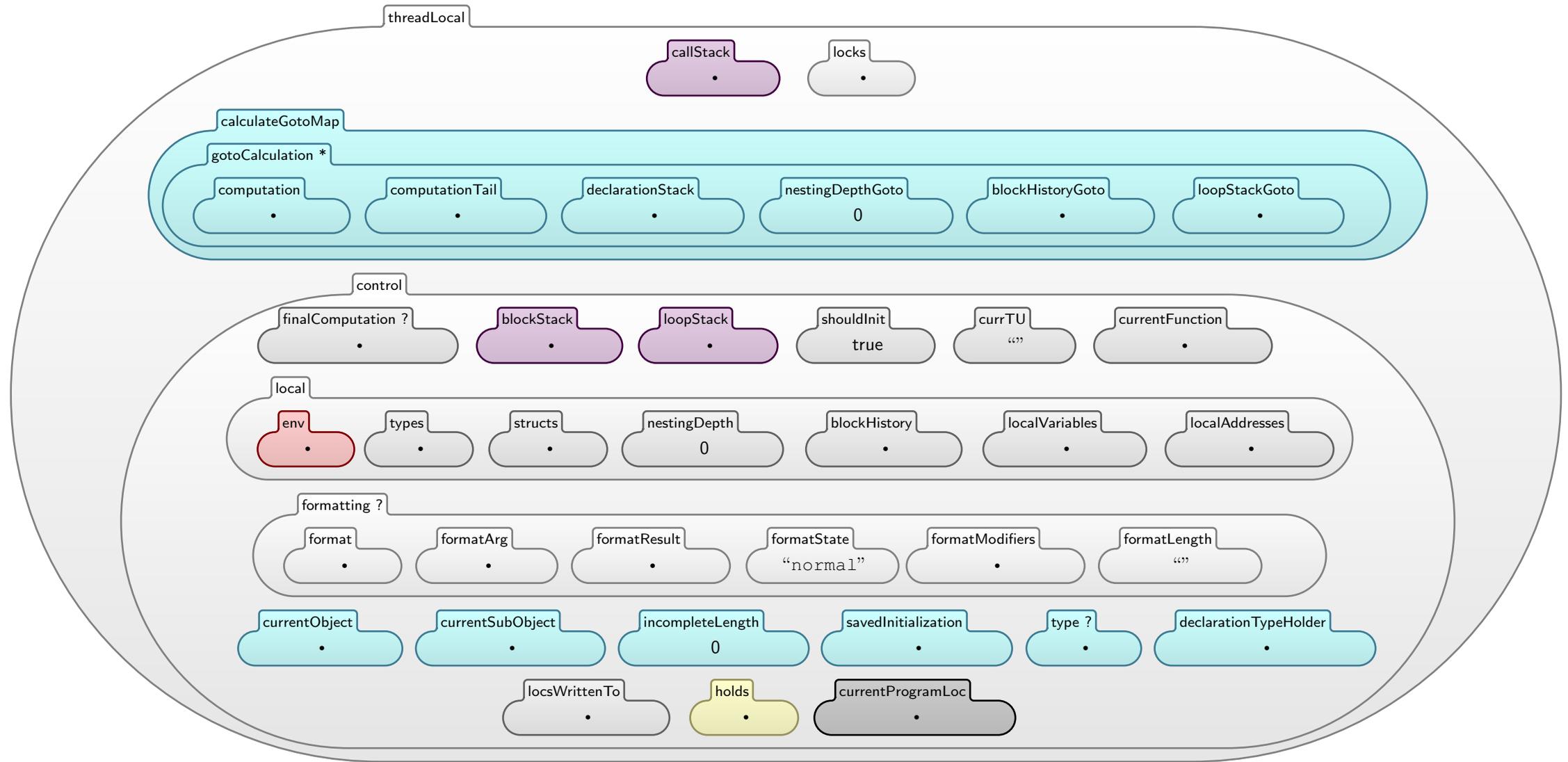
A.2 Configuration

A \mathbb{K} configuration describes the state of a running program. The C configuration has over 90 nested cells, each describing an important piece of information needed for a complete snapshot of execution.

Due to size limitations, the C configuration has been split into two parts. First, we give the outermost cells, representing global state such as memory and global types. Then we zoom into the `threadLocal` cell, which contains important information about the state of an individual thread, such as a call stack and the currently executed function.

CONFIGURATION:





A.3 Expressions

This section represents the semantics of C expressions, and generally corresponds to §6.5 in the C standard.

The smaller modules, like `COMMON-SEMANTICS-EXPRESSIONS-INCLUDE` are used for consistency. In the larger groups of modules, only a single module needs to be included instead of multiple modules that would need to be consistently updated.

```
MODULE COMMON-SEMANTICS-EXPRESSIONS-INCLUDE
```

```
  IMPORTS COMMON-INCLUDE
```

```
END MODULE
```

```
MODULE COMMON-C-EXPRESSIONS
```

```
  IMPORTS COMMON-SEMANTICS-EXPRESSIONS-INCLUDE
```

```
  RULE
```

```
    NothingExpression  
    ───────────  
    emptyValue  
    [anywhere]
```

```
END MODULE
```

```
MODULE DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE
```

```
  IMPORTS DYNAMIC-INCLUDE
```

```
  SYNTAX  $K ::= \text{assign}(K, K)$ 
```

```
  SYNTAX  $Id ::= \text{compoundLiteral}(Nat)$ 
```

```
  SYNTAX  $K ::= \text{lvGetOffset}(K, K, Type)$   
          |  $\text{tvGetOffset}(K, K, Type)$   
          |  $\text{fromArray}(Int, Nat)$   
          |  $\text{makeTruth}(Bool)$  [function]
```

```
  DEFINE
```

```
    makeTruth( $B$ )  
    ───────────  
    if  $B$   
    then 1:t(•, int)  
    else  
      0:t(•, int)  
    fi
```

```
  SYNTAX  $Nat ::= \text{arrayLength}(KResult)$  [function]
```

```

DEFINE
  arrayLength(t(—, arrayType(—, N)))
  N

```

these large numbers are used instead of an infinity—the result of these rules shouldn't be used later anyway

```

DEFINE
  arrayLength(t(—, flexibleArrayType(—)))
  36893488147419103232

```

```

DEFINE
  arrayLength(t(—, incompleteArrayType(—)))
  36893488147419103232

```

```

RULE REVAL-SKIPVAL
  reval(skipval)
  skipval

```

```

RULE REVAL-EMPTYVAL
  reval(emptyValue)
  emptyValue

```

```

RULE REVAL-TV-NORMAL
  reval(L: T)
  L: T

```

```

RULE PEVAL-TV-NORMAL
  peval(V: T)
  V: T

```

(n1570) §6.3.2.1 ¶2 Except when it is the operand of the **sizeof** operator, the `_Alignof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the `.` operator or an assignment operator, an lvalue that does not have an array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called lvalue conversion. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue. ...

```

RULE
  reval(lv(Loc, T))
  read(Loc, unqualifyType(T))
  when (¬Bool isArrayType(T)) ∧ Bool (¬Bool isFunctionType(T))

```

RULE

$$\frac{\text{peval}(\text{lv}(Loc, T))}{\text{lv}(Loc, T)}$$

when $(\neg_{Bool} \text{isArrayType}(T)) \wedge_{Bool} (\neg_{Bool} \text{isFunctionType}(T))$

(n1570) §6.3.2.1 ¶3 Except when it is the operand of the **sizeof** operator, the `_Alignof` operator, or the unary `&` operator, or is a string literal used to initialize an array, an expression that has type “array of type” is converted to an expression with type “pointer to type” that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

RULE **REVAL-LV-ARRAY**

$$\frac{\text{reval}(\text{lv}(Loc, T))}{Loc:t(\text{fromArray}(0, \text{arrayLength}(T)), \text{pointerType}(\text{innerType}(T)))}$$

when $\text{isArrayType}(T)$

RULE **PEVAL-LV-ARRAY**

$$\frac{\text{peval}(\text{lv}(Loc, T))}{Loc:t(\text{fromArray}(0, \text{arrayLength}(T)), \text{pointerType}(\text{innerType}(T)))}$$

when $\text{isArrayType}(T)$

(n1570) §6.3.2.1 ¶4 A function designator is an expression that has function type. Except when it is the operand of the **sizeof** operator, the `_Alignof` operator, or the unary `&` operator, a function designator with type “function returning T” is converted to an expression that has type “pointer to a function returning T”.

RULE **REVAL-FUNCTION**

$$\frac{\text{reval}(\text{lv}(Loc, T))}{Loc:t(\bullet, \text{pointerType}(T))}$$

when $\text{isFunctionType}(T)$

RULE **PEVAL-FUNCTION**

$$\frac{\text{peval}(\text{lv}(Loc, T))}{Loc:t(\bullet, \text{pointerType}(T))}$$

when $\text{isFunctionType}(T)$

END MODULE

MODULE DYNAMIC-SEMANTICS-LITERALS

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

SYNTAX $Bool ::= \text{withinRange}(Int, SimpleType)$ [function]

DEFINE

$$\frac{\text{withinRange}(I, T)}{(I \leq_{Int} \max(t(\bullet, T))) \wedge_{Bool} (I \geq_{Int} \min(t(\bullet, T)))}$$

SYNTAX $String ::= \text{simplifyForHex}(String)$ [function]

DEFINE

$$\frac{\text{simplifyForHex}(S)}{\text{simplifyForHex}(\text{butFirstChar}(S)) \text{ when } (\text{firstChar}(S) ==_{String} \text{"0"}) \wedge_{Bool} (\text{lengthString}(S) >_{Int} 1)}$$

DEFINE

$$\frac{\text{simplifyForHex}(S)}{S} \text{ when } \left(\text{firstChar}(S) \neq_{String} \text{"0"} \right) \vee_{Bool} (\text{lengthString}(S) ==_{Int} 1)$$

SYNTAX $K ::= \text{hexOrOctalConstant}(K)$

RULE

$$\frac{\text{HexConstant}(S)}{\text{hexOrOctalConstant}(\text{String2Rat}(\text{simplifyForHex}(S), 16))} \text{ [anywhere]}$$

RULE

$$\frac{\text{OctalConstant}(N)}{\text{hexOrOctalConstant}(\text{String2Rat}(\text{Rat2String}(N, 10), 8))} \text{ [anywhere]}$$

(n1570) §6.4.4.1 ¶4–6

The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16. The lexically first digit is the most significant.

The type of an integer constant is the first of the corresponding list in which its value can be represented.

Suffix	Decimal Constant	Octal or Hexadecimal Constant
none	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u or U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l or L	long int long long int	long int unsigned long int long long int unsigned long long int
Both u or U and l or L	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll or LL	long long int	long long int unsigned long long int
Both u or U and ll or LL	unsigned long long int	unsigned long long int

If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. If an integer constant cannot be represented by any type in its list and has no extended integer type, then the integer constant has no type.

RULE

NoSuffix(DecimalConstant(*I*))

```
if  withinRange(I,int)
then I:t(•,int)
else
  if  withinRange(I,long-int)
  then I:t(•,long-int)
  else
    if  withinRange(I,long-long-int)
    then I:t(•,long-long-int)
    else
      I:t(•,no-type)
    fi
  fi
fi
[anywhere]
```

RULE

NoSuffix(hexOrOctalConstant(*I*))

```
if  withinRange(I,int)
then I:t(•,int)
else
  if  withinRange(I,unsigned-int)
  then I:t(•,unsigned-int)
  else
    if  withinRange(I,long-int)
    then I:t(•,long-int)
    else
      if  withinRange(I,unsigned-long-int)
      then I:t(•,unsigned-long-int)
      else
        if  withinRange(I,long-long-int)
        then I:t(•,long-long-int)
        else
          if  withinRange(I,unsigned-long-long-int)
          then I:t(•,unsigned-long-long-int)
          else
            I:t(•,no-type)
          fi
        fi
      fi
    fi
  fi
fi
[anywhere]
```

RULE

$$\frac{U(\text{hexOrOctalConstant}(I))}{\begin{array}{l} \text{if } \text{withinRange}(I, \text{unsigned-int}) \\ \text{then } I:t(\bullet, \text{unsigned-int}) \\ \text{else} \\ \quad \text{if } \text{withinRange}(I, \text{unsigned-long-int}) \\ \quad \text{then } I:t(\bullet, \text{unsigned-long-int}) \\ \quad \text{else} \\ \quad \quad \text{if } \text{withinRange}(I, \text{unsigned-long-long-int}) \\ \quad \quad \text{then } I:t(\bullet, \text{unsigned-long-long-int}) \\ \quad \quad \text{else} \\ \quad \quad \quad I:t(\bullet, \text{no-type}) \\ \quad \quad \text{fi} \\ \quad \text{fi} \\ \text{fi} \\ \text{[anywhere]} \end{array}}$$

RULE

$$\frac{L(\text{hexOrOctalConstant}(I))}{\begin{array}{l} \text{if } \text{withinRange}(I, \text{long-int}) \\ \text{then } I:t(\bullet, \text{long-int}) \\ \text{else} \\ \quad \text{if } \text{withinRange}(I, \text{unsigned-long-int}) \\ \quad \text{then } I:t(\bullet, \text{unsigned-long-int}) \\ \quad \text{else} \\ \quad \quad \text{if } \text{withinRange}(I, \text{long-long-int}) \\ \quad \quad \text{then } I:t(\bullet, \text{long-long-int}) \\ \quad \quad \text{else} \\ \quad \quad \quad \text{if } \text{withinRange}(I, \text{unsigned-long-long-int}) \\ \quad \quad \quad \text{then } I:t(\bullet, \text{unsigned-long-long-int}) \\ \quad \quad \quad \text{else} \\ \quad \quad \quad \quad I:t(\bullet, \text{no-type}) \\ \quad \quad \quad \text{fi} \\ \quad \quad \text{fi} \\ \quad \text{fi} \\ \text{fi} \\ \text{[anywhere]} \end{array}}$$

```

RULE
      UL(hexOrOctalConstant(I))
-----
if   withinRange(I, unsigned-long-int)
then I:t(•, unsigned-long-int)
else
    if   withinRange(I, unsigned-long-long-int)
    then I:t(•, unsigned-long-long-int)
    else
        I:t(•, no-type)
    fi
fi
[anywhere]

```

```

RULE
      LL(hexOrOctalConstant(I))
-----
if   withinRange(I, long-long-int)
then I:t(•, long-long-int)
else
    if   withinRange(I, unsigned-long-long-int)
    then I:t(•, unsigned-long-long-int)
    else
        I:t(•, no-type)
    fi
fi
[anywhere]

```

```

RULE
      ULL(hexOrOctalConstant(I))
-----
if   withinRange(I, unsigned-long-long-int)
then I:t(•, unsigned-long-long-int)
else
    I:t(•, no-type)
fi
[anywhere]

```

RULE

$$\frac{U(\text{DecimalConstant}(I))}{\begin{array}{l} \text{if } \text{withinRange}(I, \text{unsigned-int}) \\ \text{then } I:t(\bullet, \text{unsigned-int}) \\ \text{else} \\ \quad \text{if } \text{withinRange}(I, \text{unsigned-long-int}) \\ \quad \text{then } I:t(\bullet, \text{unsigned-long-int}) \\ \quad \text{else} \\ \quad \quad \text{if } \text{withinRange}(I, \text{unsigned-long-long-int}) \\ \quad \quad \text{then } I:t(\bullet, \text{unsigned-long-long-int}) \\ \quad \quad \text{else} \\ \quad \quad \quad I:t(\bullet, \text{no-type}) \\ \quad \quad \text{fi} \\ \quad \text{fi} \\ \text{fi} \\ \text{[anywhere]} \end{array}}$$

RULE

$$\frac{L(\text{DecimalConstant}(I))}{\begin{array}{l} \text{if } \text{withinRange}(I, \text{long-int}) \\ \text{then } I:t(\bullet, \text{long-int}) \\ \text{else} \\ \quad \text{if } \text{withinRange}(I, \text{long-long-int}) \\ \quad \text{then } I:t(\bullet, \text{long-long-int}) \\ \quad \text{else} \\ \quad \quad I:t(\bullet, \text{no-type}) \\ \quad \text{fi} \\ \text{fi} \\ \text{[anywhere]} \end{array}}$$

RULE

$$\frac{\text{UL(DecimalConstant}(I))}{\text{if } \text{withinRange}(I, \text{unsigned-long-int}) \\ \text{then } I:t(\bullet, \text{unsigned-long-int}) \\ \text{else} \\ \quad \text{if } \text{withinRange}(I, \text{unsigned-long-long-int}) \\ \quad \text{then } I:t(\bullet, \text{unsigned-long-long-int}) \\ \quad \text{else} \\ \quad \quad I:t(\bullet, \text{no-type}) \\ \quad \text{fi} \\ \text{fi} \\ \text{[anywhere]}}$$

RULE

$$\frac{\text{LL(DecimalConstant}(I))}{\text{if } \text{withinRange}(I, \text{long-long-int}) \\ \text{then } I:t(\bullet, \text{long-long-int}) \\ \text{else} \\ \quad I:t(\bullet, \text{no-type}) \\ \text{fi} \\ \text{[anywhere]}}$$

RULE

$$\frac{\text{ULL(DecimalConstant}(I))}{\text{if } \text{withinRange}(I, \text{unsigned-long-long-int}) \\ \text{then } I:t(\bullet, \text{unsigned-long-long-int}) \\ \text{else} \\ \quad I:t(\bullet, \text{no-type}) \\ \text{fi} \\ \text{[anywhere]}}$$

(n1570) §6.4.4.2 ¶4 An unsuffixed floating constant has type **double**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**.

SYNTAX $K ::= \text{reducedFloat}(Float)$

RULE

$$\frac{\text{DecimalFloatConstant}(\text{---}, \text{---}, F)}{\text{reducedFloat}(F)}$$
 [anywhere]

RULE

$$\frac{\text{HexFloatConstant}(\text{---}, \text{---}, F)}{\text{reducedFloat}(F)}$$
 [anywhere]

RULE

$$\frac{\text{NoSuffix}(\text{reducedFloat}(F))}{F : \text{t}(\bullet, \text{double})}$$
 [anywhere]

RULE

$$\frac{\text{L}(\text{reducedFloat}(F))}{F : \text{t}(\bullet, \text{long-double})}$$
 [anywhere]

RULE

$$\frac{\text{F}(\text{reducedFloat}(F))}{F : \text{t}(\bullet, \text{float})}$$
 [anywhere]

(n1570) §6.4.4.4 ¶10 An integer character constant has type **int**. The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character (e.g., 'ab'), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of the single character or escape sequence is converted to type **int**.

RULE

$$\frac{\text{CharLiteral}(N)}{\text{cast}(\text{t}(\bullet, \text{int}), \text{cast}(\text{t}(\bullet, \text{char}), N : \text{t}(\bullet, \text{int})))}$$
 [anywhere]

RULE

$$\frac{\text{WCharLiteral}(N)}{N : \text{cfg:wcharut}}$$
 [anywhere]

RULE

$$\frac{\text{Constant}(V)}{V}$$
 [anywhere]

RULE CREATE-INTERNAL-VALUE

$$\frac{N}{N : \text{cfg:largestUnsigned}}$$

(n1570) §6.4.5 ¶6 For character string literals, the array elements have type **char**, and are initialized with the individual bytes of the multibyte character sequence. ... For wide string literals prefixed by the letter L, the array elements have type `wchar_t` and are initialized with the sequence of wide characters corresponding to the multibyte character sequence...

RULE CONST-STRING-NOTFOUND

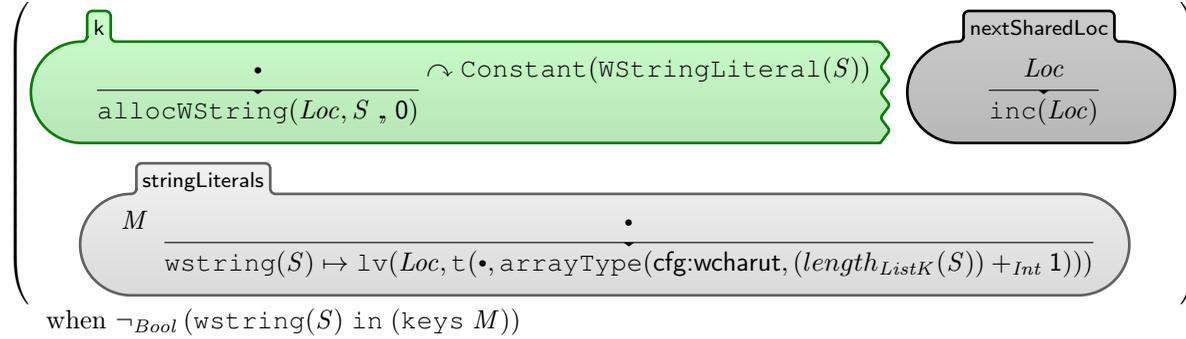
$$\left(\frac{\frac{\cdot}{\text{allocString}(Loc, S + \text{String} \text{ "\000"})} \rightsquigarrow \text{Constant}(\text{StringLiteral}(S))}{\text{stringLiterals}} \quad \frac{Loc}{\text{inc}(Loc)} \right)$$

$$\frac{M}{S \mapsto \text{lv}(Loc, \text{t}(\cdot, \text{arrayType}(\text{t}(\cdot, \text{char}), \text{lengthString}(S) + \text{Int } 1))}$$

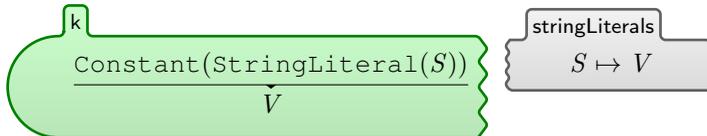
when $\neg_{\text{Bool}}(S \text{ in } (\text{keys } M))$

SYNTAX $K ::= \text{wstring}(\text{List}\{K\})$

RULE **CONST-WSTRING-NOTFOUND**



RULE **CONST-STRING-FOUND**



RULE **CONST-WSTRING-FOUND**



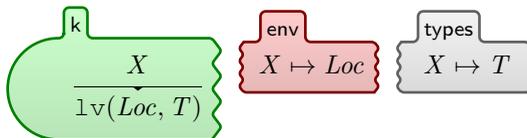
END MODULE

MODULE DYNAMIC-SEMANTICS-IDENTIFIERS

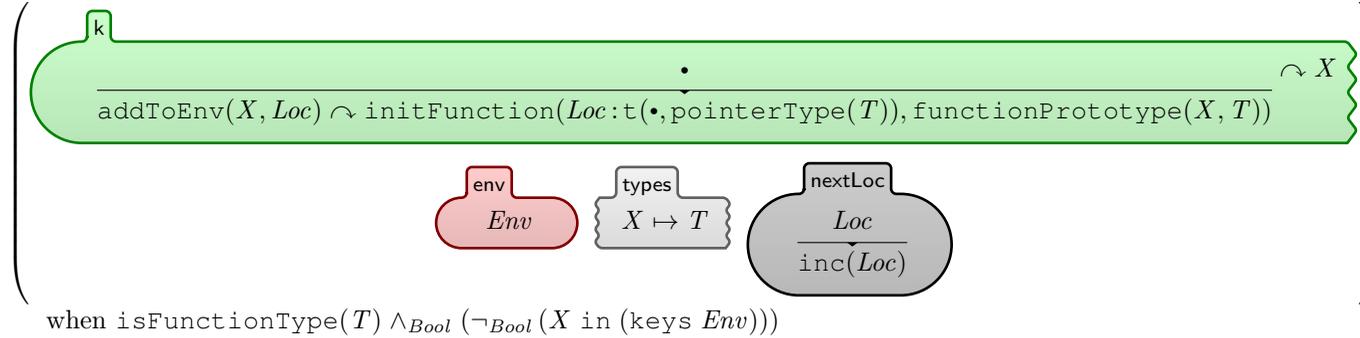
IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.1 ¶2 An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).

RULE **LOOKUP**



RULE LOOKUP-BUILTIN-FUNCTION-NOTFOUND



END MODULE

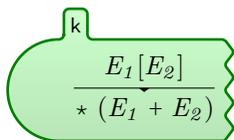
MODULE DYNAMIC-SEMANTICS-ARRAY-SUBSCRIPTING

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.2.1 ¶2-3 A postfix expression followed by an expression in square brackets [] is a subscripted designation of an element of an array object. The definition of the subscript operator [] is that $E_1[E_2]$ is identical to $(*((E_1) + (E_2)))$. Because of the conversion rules that apply to the binary + operator, if E_1 is an array object (equivalently, a pointer to the initial element of an array object) and E_2 is an integer, $E_1[E_2]$ designates the E_2 -th element of E_1 (counting from zero).

Successive subscript operators designate an element of a multidimensional array object. If E is an n -dimensional array ($n \geq 2$) with dimensions $i \times j \times \dots \times k$, then E (used as other than an lvalue) is converted to a pointer to an $(n - 1)$ -dimensional array with dimensions $j \times \dots \times k$. If the unary * operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced $(n - 1)$ -dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

RULE ARRAY-SUBSCRIPT



END MODULE

MODULE DYNAMIC-SEMANTICS-FUNCTION-CALLS

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

SYNTAX $K ::= \text{application}(K, \text{List}\{K\text{Result}\}) \text{ [strict(1)]}$
 $\quad \quad \quad | \text{application}'(K, \text{List}\{K\text{Result}\}) \text{ [strict(1)]}$

(n1570) §6.5.2.2 ¶3 A postfix expression followed by parentheses () containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

(n1570) §6.5.2.2 ¶6 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **float** are promoted to **double**. These are called the default argument promotions. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (, . . .) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

- one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types;
- both types are pointers to qualified or unqualified versions of a character type or **void**.

RULE **FUNCTION-APPLICATION-PRE**

$$\frac{\text{Call}(\text{Loc}:\text{t}(-, \text{pointerType}(T)), \text{List}(L))}{\text{application}(\text{readFunction}(\text{Loc}), L)}$$

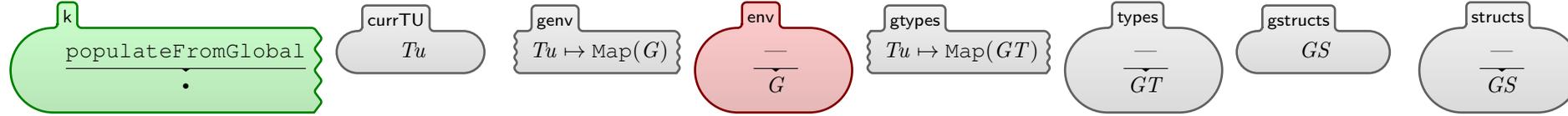
when $\text{isFunctionType}(T)$

this extra step is useful for putting the function name in the transition graph

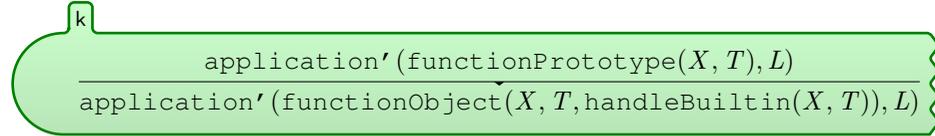
RULE **CALL**

$$\frac{\text{application}(\text{Fun}, L)}{\text{application}'(\text{Fun}, L)}$$

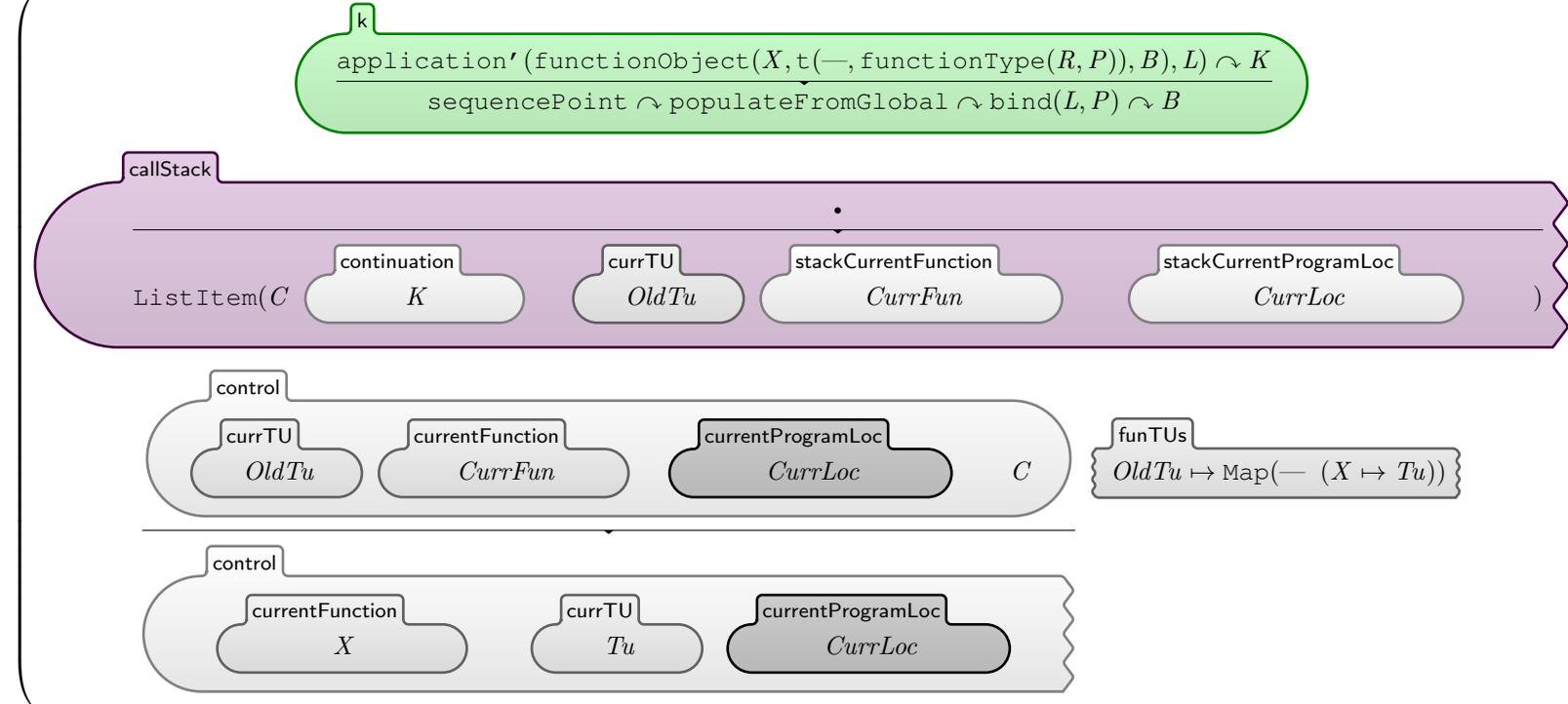
RULE POPULATEFROMGLOBAL



RULE BUILTIN-APPLICATION



RULE FUNCTION-APPLICATION



END MODULE

MODULE DYNAMIC-SEMANTICS-MEMBERS

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.2.3 ¶3 A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

RULE

$$\frac{\text{lv}(Loc, \text{t}(_, \text{structType}(S))) \cdot F}{\text{lv}(Loc + \text{bits } Offset, T)} \quad \text{structs } S \mapsto \text{aggregateInfo}(_, _ (F \mapsto T), _ (F \mapsto Offset))$$

RULE

$$\frac{\text{lv}(Loc, \text{t}(_, \text{unionType}(S))) \cdot F}{\text{lv}(Loc + \text{bits } Offset, \text{t}(\text{fromUnion}(S) \text{ } Se, T))} \quad \text{structs } S \mapsto \text{aggregateInfo}(_, _ (F \mapsto \text{t}(Se, T)), _ (F \mapsto Offset))$$

RULE

$$\frac{L : T \cdot F}{\text{extractField}(L, T, F)}$$

(n1570) §6.5.2.3 ¶4 A postfix expression followed by the `->` operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue. If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

MACRO

$K \rightarrow F = (* K) \cdot F$

END MODULE

MODULE DYNAMIC-SEMANTICS-POSTFIX-INCREMENT-AND-DECREMENT

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

SYNTAX $K ::= \text{postOpRef}(K, KLabel)$
 $\quad | \text{postInc}(K, K, Type) [\text{strict}(2)]$
 $\quad | \text{postDec}(K, K, Type) [\text{strict}(2)]$

(n1570) §6.5.2.4 ¶2 The result of the postfix ++ operator is the value of the operand. As a side effect, the value of the operand object is incremented (that is, the value 1 of the appropriate type is added to it). See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. The value computation of the result is sequenced before the side effect of updating the stored value of the operand. With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation. Postfix ++ on an object with atomic type is a read-modify-write operation with `memory_order_seq_cst` memory order semantics.

RULE POST-INCREMENT-START

$$\frac{\text{kv} \quad \text{lv}(Loc, T) ++}{\text{postInc}(Loc, \text{read}(Loc, T), T)}$$

RULE POST-INCREMENT

$$\frac{\text{kv} \quad \text{postInc}(Loc, V : T, T)}{(\text{lv}(Loc, T) := (V : T + 1 : \text{t}(\bullet, \text{int}))) \curvearrowright \text{discard} \curvearrowright V : T}$$

(n1570) §6.5.2.4 ¶3 The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

RULE POST-DECREMENT-START

$$\frac{\text{kv} \quad \text{lv}(Loc, T) --}{\text{postDec}(Loc, \text{read}(Loc, T), T)}$$

RULE POST-DECREMENT

$$\frac{\text{kv} \quad \text{postDec}(Loc, V : T, T)}{(\text{lv}(Loc, T) := (V : T - 1 : \text{t}(\bullet, \text{int}))) \curvearrowright \text{discard} \curvearrowright V : T}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-COMPOUND-LITERAL

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

SYNTAX $K ::= \text{handleCompoundLiteral}(K) \text{ [strict]}$

(n1570) §6.5.2.5 ¶3–7 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose value is given by the initializer list.

If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.9, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block.

All the semantic rules for initializer lists in 6.7.9 also apply to compound literals.

String literals, and compound literals with const-qualified types, need not designate distinct objects.

We use `compoundLiteral(N:Nat)` here as the identifier of the compound literal.

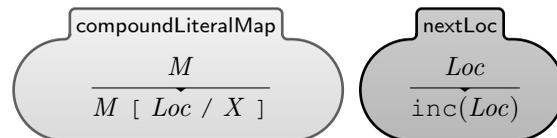
RULE

$$\frac{\text{CompoundLiteral}(N, T, K, \text{Init})}{\text{handleCompoundLiteral}(\text{figureInit}(\text{compoundLiteral}(N), \text{DeclType}(T, K), \text{Init}))}$$

RULE

$$\frac{\text{handleCompoundLiteral}(\text{initValue}(X, T, \text{Init}))}{\text{allocateType}(Loc, T) \curvearrow \text{addToEnv}(X, Loc) \curvearrow \text{giveType}(X, T) \curvearrow \text{initialize}(X, T, \text{Init}) \curvearrow X}$$

when $\neg_{Bool}(X \text{ in } (\text{keys } M))$



RULE

$$\frac{\text{handleCompoundLiteral}(\text{initValue}(X, T, \text{Init}))}{\text{addToEnv}(X, Loc) \curvearrow \text{giveType}(X, T) \curvearrow \text{initialize}(X, T, \text{Init}) \curvearrow X}$$

`compoundLiteralMap`
 $X \mapsto Loc$

END MODULE

MODULE DYNAMIC-SEMANTICS-PREFIX-INCREMENT-AND-DECREMENT

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.3.1 ¶2 The value of the operand of the prefix ++ operator is incremented. The result is the new value of the operand after incrementation. The expression ++E is equivalent to (E+=1). See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

RULE

$$\frac{++ E}{E += 1 : t(\bullet, \text{int})}$$

(n1570) §6.5.3.1 ¶3 The prefix -- operator is analogous to the prefix ++ operator, except that the value of the operand is decremented.

RULE

$$\frac{-- E}{E -= 1 : t(\bullet, \text{int})}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-REFERENCE

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.3.2 ¶3 The unary & operator yields the address of its operand. If the operand has type “type”, the result has type “pointer to type”. If the operand is the result of a unary * operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a [] operator, neither the & operator nor the unary * that is implied by the [] is evaluated and the result is as if the & operator were removed and the [] operator were changed to a + operator. Otherwise, the result is a pointer to the object or function designated by its operand.

RULE REF

$$\frac{\& l_v(\text{Loc}, T)}{\text{Loc} : t(\bullet, \text{pointerType}(T))}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-DEREFERENCE

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.3.2 ¶4 The unary $*$ operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type “pointer to type”, the result has type “type”. If an invalid value has been assigned to the pointer, the behavior of the unary $*$ operator is undefined.

RULE Deref

$$\frac{\text{checkDerefLoc}(Loc) \rightsquigarrow lv(Loc, t(S, T))}{* Loc : t(-, \text{pointerType}(t(S, T)))}$$

when $\neg_{Bool}(T ==_K \text{void})$

END MODULE

MODULE DYNAMIC-SEMANTICS-UNARY-ARITHMETIC

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.3.3 ¶2 The result of the unary $+$ operator is the value of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

RULE UNARYPLUS-INT

$$\frac{+ I : T}{\text{arithInterpret}(T, I)}$$

when $\text{isPromoted}(T)$

RULE UNARYPLUS-FLOAT

$$\frac{+ F : T}{F : T}$$

(n1570) §6.5.3.3 ¶3 The result of the unary $-$ operator is the negative of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

RULE **UNARYMINUS-INT**

$$\frac{- I : T}{\text{arithInterpret}(T, 0 -_{Int} I)} \\ \text{when isPromoted}(T)$$

RULE **UNARYMINUS-FLOAT**

$$\frac{- F : T}{\text{arithInterpret}(T, 0.0 -_{Float} F)}$$

(n1570) §6.5.3.3 ¶4 The result of the `~` operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integer promotions are performed on the operand, and the result has the promoted type. If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`.

RULE

$$\frac{\sim I : T}{\text{arithInterpret}(T, \sim_{Int} I)} \\ \text{when isPromoted}(T)$$

(n1570) §6.5.3.3 ¶5 The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type `int`. The expression `!E` is equivalent to `(0==E)`.

RULE

$$\frac{! E}{0 : \text{t}(\cdot, \text{int}) == E}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-SIZEOF

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.3.4 ¶2 The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

(n1570) §6.5.3.4 ¶5 The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is `size_t`, defined in `<stddef.h>` (and other headers).

RULE

$$\frac{\text{sizeofType}(T)}{\text{cast}(\text{cfg}:\text{sizeut}, \text{byteSizeofType}(T))}$$

SYNTAX $K ::= \text{byteSizeofType-aux}(K)$ [strict]

RULE

$$\frac{\text{byteSizeofType}(T)}{\text{byteSizeofType-aux}(\text{bitSizeofType}(T))}$$

RULE

$$\frac{\text{byteSizeofType-aux}(N : T)}{\text{bitsToBytes}(N) : T}$$

RULE

$$\frac{\text{sizeofExpression}(E)}{\text{sizeofType}(\text{typeof}(E))}$$

RULE

$$\frac{\text{sizeofType}(T, K)}{\text{sizeofType}(\text{DeclType}(T, K))}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-CAST

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.4 ¶5–6 Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a cast. A cast that specifies no conversion has no effect on the type or value of an expression.

If the value of the expression is represented with greater range or precision than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type and removes any extra range and precision.

RULE

$$\frac{\text{Cast}(T, K, V)}{\text{cast}(\text{DeclType}(T, K), V)}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-MULTIPLICATIVE-OPERATORS

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.5 ¶3–6 The usual arithmetic conversions are performed on the operands.

The result of the binary $*$ operator is the product of the operands.

The result of the $/$ operator is the quotient from the division of the first operand by the second; the result of the $\%$ operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

When integers are divided, the result of the $/$ operator is the algebraic quotient with any fractional part discarded. If the quotient a/b is representable, the expression $(a/b) * b + a \% b$ shall equal a ; otherwise, the behavior of both a/b and $a \% b$ is undefined.

RULE

$$\frac{I_1 : T * I_2 : T}{\text{arithInterpret}(T, I_1 *_{Int} I_2)}$$

when isPromoted(T)

RULE

$$\frac{F_1 : T * F_2 : T}{\text{arithInterpret}(T, F_1 *_{Float} F_2)}$$

RULE

$$\frac{I_1 : T / I_2 : T}{\text{arithInterpret}(T, I_1 \div_{Int} I_2)}$$

when isPromoted(T) $\wedge_{Bool} (I_2 \neq_{Int} 0)$

RULE

$$\frac{F_1 : T / F_2 : T}{\text{arithInterpret}(T, F_1 \div_{Float} F_2)}$$

RULE

$$\frac{\text{piece}(\text{unknown}(N), N) \div_{Int} M}{\text{piece}(\text{unknown}(N), N)}$$

when $(M \neq_{Int} 0) \wedge_{Bool} \text{isConcreteNumber}(M)$
[anywhere]

RULE

$$\frac{I_1 : T \ \% \ I_2 : T}{\text{arithInterpret}(T, I_1 \%_{Int} I_2)}$$

when $((\text{isPromoted}(T) \wedge_{Bool} (\min(T) \leq_{Int} (I_1 \div_{Int} I_2))) \wedge_{Bool} (\max(T) \geq_{Int} (I_1 \div_{Int} I_2))) \wedge_{Bool} (I_2 \neq_{Int} 0)$

END MODULE

MODULE DYNAMIC-SEMANTICS-ADDITIVE-OPERATORS

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.6 ¶2 For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type.

(n1570) §6.5.6 ¶3 For subtraction, one of the following shall hold:

- both operands have arithmetic type;
- both operands are pointers to qualified or unqualified versions of compatible complete object types; or
- the left operand is a pointer to a complete object type and the right operand has integer type.

(Decrementing is equivalent to subtracting 1.)

(n1570) §6.5.6 ¶4 If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

(n1570) §6.5.6 ¶5 The result of the binary + operator is the sum of the operands.

(n1570) §6.5.6 ¶6 The result of the binary – operator is the difference resulting from the subtraction of the second operand from the first.

(n1570) §6.5.6 ¶7 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

(n1570) §6.5.6 ¶8 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression P points to the i -th element of an array object, the expressions $(P) + N$ (equivalently, $N + (P)$) and $(P) - N$ (where N has the value n) point to, respectively, the $i + n$ -th and $i - n$ -th elements of the array object, provided they exist. Moreover, if the expression P points to the last element of an array object, the expression $(P) + 1$ points one past the last element of the array object, and if the expression Q points one past the last element of an array object, the expression $(Q) - 1$ points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated.

SYNTAX $K ::= \text{addToPointer}(K, \text{Type}, K, K)$ [strict(4)]

RULE

k

$$\frac{Loc : \text{t}(S, \text{pointerType}(T')) + I : T}{\text{addToPointer}(Loc, \text{t}(S, \text{pointerType}(T')), I, \text{sizeofType}(T'))}$$

when $\text{hasIntegerType}(T) \wedge_{Bool} (T' \neq_K \text{void})$

RULE

$$\frac{\text{k} \quad I : T + Loc : \tau(S, \text{pointerType}(T'))}{\text{addToPointer}(Loc, \tau(S, \text{pointerType}(T')), I, \text{sizeofType}(T'))}$$

when $\text{hasIntegerType}(T) \wedge_{Bool} (T' \neq_K \text{void})$

RULE

$$\frac{\text{k} \quad Loc : \tau(S, \text{pointerType}(T')) - I : T}{\text{addToPointer}(Loc, \tau(S, \text{pointerType}(T')), 0 -_{Int} I, \text{sizeofType}(T'))}$$

when $\text{hasIntegerType}(T) \wedge_{Bool} (T' \neq_K \text{void})$

RULE

$$\frac{\text{k} \quad \text{addToPointer}(Loc, T, I, Size : -)}{Loc +_{Int} (I *_{Int} Size) : \text{newFromArray}(T, I)}$$

when $\text{ifFromArrayInBounds}(T, I)$

SYNTAX $KResult ::= \text{newFromArray}(KResult, Int)$ [function]

DEFINE

$$\frac{\text{newFromArray}(\tau(\text{fromArray}(Offset, Len), \text{pointerType}(T)), I)}{\tau(\text{fromArray}(Offset +_{Int} I, Len), \text{pointerType}(T))}$$

DEFINE

$$\frac{\text{newFromArray}(\tau(\bullet, \text{pointerType}(T)), I)}{\tau(\bullet, \text{pointerType}(T))}$$

SYNTAX $Bool ::= \text{ifFromArrayInBounds}(KResult, Int)$ [function]

DEFINE

$$\frac{\text{ifFromArrayInBounds}(\tau(\text{fromArray}(Offset, Len), \text{pointerType}(T)), I)}{\text{true}}$$

when $(Offset +_{Int} I) \leq_{Int} Len$

DEFINE

$$\frac{\text{ifFromArrayInBounds}(t(\text{fromArray}(Offset, Len), \text{pointerType}(T)), I)}{\text{false}}$$
 when $(Offset +_{Int} I) >_{Int} Len$

DEFINE

$$\frac{\text{ifFromArrayInBounds}(t(\bullet, \text{pointerType}(T)), -)}{\text{true}}$$

(n1570) §6.5.6 ¶9 When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` defined in the `<stddef.h>` header. If the result is not representable in an object of that type, the behavior is undefined. In other words, if the expressions P and Q point to, respectively, the i -th and j -th elements of an array object, the expression $(P) - (Q)$ has the value $i - j$ provided the value fits in an object of type `ptrdiff_t`. Moreover, if the expression P points either to an element of an array object or one past the last element of an array object, and the expression Q points to the last element of the same array object, the expression $((Q) + 1) - (P)$ has the same value as $((Q) - (P)) + 1$ and as $-(P) - ((Q) + 1)$, and has the value zero if the expression P points one past the last element of the array object, even though the expression $(Q) + 1$ does not point to an element of the array object.

SYNTAX $K ::= \text{computePointerDifference}(Int, Int, K)$ [strict(3)]

RULE **START-POINTER-DIFFERENCE**

$$\frac{I_1 : t(-, \text{pointerType}(T)) - I_2 : t(-, \text{pointerType}(T))}{\text{computePointerDifference}(I_1, I_2, \text{sizeofType}(T))}$$

RULE **POINTER-DIFFERENCE**

$$\frac{\text{computePointerDifference}(\text{loc}(Base, Offset_1, 0), \text{loc}(Base, Offset_2, 0), Size : -)}{\text{when } ((Offset_1 -_{Int} Offset_2) \div_{Int} Size : \text{cfg:ptrdiffut}) ==_{Int} 0}$$

RULE

$$\frac{I_1 : T + I_2 : T}{\text{arithInterpret}(T, I_1 +_{Int} I_2)}$$

when `isPromoted(T)`

RULE

$$\frac{I_1 : T - I_2 : T}{\text{arithInterpret}(T, I_1 -_{Int} I_2)}$$

when `isPromoted(T)`

RULE

$$\frac{F_1 : T + F_2 : T}{\text{arithInterpret}(T, F_1 +_{\text{Float}} F_2)}$$

RULE

$$\frac{F_1 : T - F_2 : T}{\text{arithInterpret}(T, F_1 -_{\text{Float}} F_2)}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-BITWISE-SHIFT

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

SYNTAX $K ::= \text{leftShiftInterpret}(Type, BaseValue, K)$ [function]
| $\text{rightShiftInterpret}(Type, BaseValue)$ [function]

(n1570) §6.5.7 ¶3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined

(n1570) §6.5.7 ¶4 The result of $E1 \ll E2$ is $E1$ left-shifted $E2$ bit positions; vacated bits are filled with zeros. If $E1$ has an unsigned type, the value of the result is $E1 \times 2^{E2}$, reduced modulo one more than the maximum value representable in the result type. If $E1$ has a signed type and nonnegative value, and $E1 \times 2^{E2}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

RULE

$$\frac{I : T \ll N : T'}{\text{leftShiftInterpret}(T, I \ll_{\text{Int}} N, I : T)} \\ \text{when } (\text{isPromoted}(T) \wedge_{\text{Bool}} \text{isPromoted}(T')) \wedge_{\text{Bool}} (N <_{\text{Int}} \text{numBits}(T))$$

DEFINE

$$\frac{\text{leftShiftInterpret}(T, I, E_1 : T)}{I \%_{\text{Int}} (\max(T) +_{\text{Int}} 1) : T} \\ \text{when } \text{hasUnsignedIntegerType}(T)$$

DEFINE

$\frac{\text{leftShiftInterpret}(T, I, E_1 : T)}{I : T}$

when (hasSignedIntegerType(T) \wedge_{Bool} ($I \leq_{Int} \max(T)$)) \wedge_{Bool} ($I \geq_{Int} \min(T)$)

(n1570) §6.5.7 ¶5 The result of $E_1 \gg E_2$ is E_1 right-shifted E_2 bit positions. If E_1 has an unsigned type or if E_1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $E_1/2^{E_2}$. If E_1 has a signed type and a negative value, the resulting value is implementation-defined.

RULE

$\frac{I : T \gg N : T'}{\text{rightShiftInterpret}(T, I \gg_{Int} N)}$

when (isPromoted(T) \wedge_{Bool} isPromoted(T')) \wedge_{Bool} ($N <_{Int} \text{numBits}(T)$)

DEFINE

$\frac{\text{rightShiftInterpret}(T, I)}{I : T}$

when hasIntegerType(T)

END MODULE

MODULE DYNAMIC-SEMANTICS-RELATIONAL

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.8 ¶3 If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

(n1570) §6.5.8 ¶4 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

(n1570) §6.5.8 ¶5 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression P points to an element of an array object and the expression Q points to the last element of the same array object, the pointer expression $Q+1$ compares greater than P . In all other cases, the behavior is undefined.

(n1570) §6.5.8 ¶6 Each of the operators $<$ (less than), $>$ (greater than), $<=$ (less than or equal to), and $>=$ (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false. The result has type **int**.

RULE

$$\frac{I_1 : T < I_2 : T}{\text{makeTruth}(I_1 <_{Int} I_2)}$$

when $\bigvee_{Bool} ((\text{isPointerType}(T) \wedge_{Bool} \text{isConcreteNumber}(I_1)) \wedge_{Bool} \text{isConcreteNumber}(I_2))$ isPromoted(T)

RULE

$$\frac{I_1 : T \leq I_2 : T}{\text{makeTruth}(I_1 \leq_{Int} I_2)}$$

when $\bigvee_{Bool} ((\text{isPointerType}(T) \wedge_{Bool} \text{isConcreteNumber}(I_1)) \wedge_{Bool} \text{isConcreteNumber}(I_2))$ isPromoted(T)

RULE

$$\frac{I_1 : T > I_2 : T}{\text{makeTruth}(I_1 >_{Int} I_2)}$$

when $\bigvee_{Bool} ((\text{isPointerType}(T) \wedge_{Bool} \text{isConcreteNumber}(I_1)) \wedge_{Bool} \text{isConcreteNumber}(I_2))$ isPromoted(T)

RULE

$$\frac{I_1 : T \geq I_2 : T}{\text{makeTruth}(I_1 \geq_{Int} I_2)}$$

when $\bigvee_{Bool} ((\text{isPointerType}(T) \wedge_{Bool} \text{isConcreteNumber}(I_1)) \wedge_{Bool} \text{isConcreteNumber}(I_2))$ isPromoted(T)

RULE

$$\frac{F_1 : T < F_2 : T}{\text{makeTruth}(F_1 <_{Float} F_2)}$$

RULE

$$\frac{F_1 : T \leq F_2 : T}{\text{makeTruth}(F_1 \leq_{Float} F_2)}$$

RULE

$$\frac{F_1 : T > F_2 : T}{\text{makeTruth}(F_1 >_{Float} F_2)}$$

RULE

$$\frac{F_1 : T \geq F_2 : T}{\text{makeTruth}(F_1 \geq_{Float} F_2)}$$

RULE **PTR-COMPARE-LT**

$$\frac{\text{loc}(Base, Offset, 0) : T < \text{loc}(Base, Offset', 0) : T'}{\text{makeTruth}(Offset <_{Int} Offset')}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE **PTR-COMPARE-LTE**

$$\frac{\text{loc}(Base, Offset, 0) : T \leq \text{loc}(Base, Offset', 0) : T'}{\text{makeTruth}(Offset \leq_{Int} Offset')}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE **PTR-COMPARE-GT**

$$\frac{\text{loc}(Base, Offset, 0) : T > \text{loc}(Base, Offset', 0) : T'}{\text{makeTruth}(Offset >_{Int} Offset')}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE **PTR-COMPARE-GTE**

$$\frac{\text{loc}(Base, Offset, 0) : T \geq \text{loc}(Base, Offset', 0) : T'}{\text{makeTruth}(Offset \geq_{Int} Offset')}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

END MODULE

MODULE DYNAMIC-SEMANTICS-EQUALITY

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.9 ¶3–4 The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence. Each of the operators yields 1 if the specified relation is true and 0 if it is false. The result has type **int**. For any pair of operands, exactly one of the relations is true.

If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal.

RULE

$$\frac{I_1 : T == I_2 : T}{\text{makeTruth}(I_1 ==_K I_2)}$$

when $\left(\bigvee_{Bool} ((\text{isPointerType}(T) \wedge_{Bool} \text{isConcreteNumber}(I_1)) \wedge_{Bool} \text{isConcreteNumber}(I_2)) \right) \wedge_{Bool} \left(\neg_{Bool} \left(\bigvee_{Bool} \text{isUnknown}(I_2) \right) \right)$

RULE

$$\frac{I_1 : T \neq I_2 : T}{\text{makeTruth}(I_1 \neq_K I_2)}$$

when $\left(\bigvee_{Bool} ((\text{isPointerType}(T) \wedge_{Bool} \text{isConcreteNumber}(I_1)) \wedge_{Bool} \text{isConcreteNumber}(I_2)) \right) \wedge_{Bool} \left(\neg_{Bool} \left(\bigvee_{Bool} \text{isUnknown}(I_1) \right) \right)$

RULE

$$\frac{F_1 : T == F_2 : T}{\text{makeTruth}(F_1 ==_{Float} F_2)}$$

RULE

$$\frac{F_1 : T \neq F_2 : T}{\text{makeTruth}(F_1 \neq_{Float} F_2)}$$

(n1570) §6.5.9 ¶15–7 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter.

Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.

For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

RULE

$$\frac{N : T == N : T'}{\text{1: t}(\bullet, \text{int})}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE

$$\frac{N : T \neq N : T'}{\text{0: t}(\bullet, \text{int})}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE

$$\frac{\text{NullPointer} : T == N : T'}{\text{makeTruth}(\text{NullPointer} ==_K N)}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE

$$\frac{\text{NullPointer} : T \neq N : T'}{\text{makeTruth}(\text{NullPointer} \neq_K N)}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE

$$\frac{N : T == \text{NullPointer} : T'}{\text{makeTruth}(\text{NullPointer} ==_K N)}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE

$$\frac{N : T \neq \text{NullPointer} : T'}{\text{makeTruth}(\text{NullPointer} \neq_K N)}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE

k

$$\frac{\text{loc}(Base, Offset, 0) : T == \text{loc}(Base, Offset', 0) : T'}{\text{makeTruth}(Offset ==_{Int} Offset')}$$

when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE

k

$$\frac{\text{loc}(Base, Offset, 0) : T \neq \text{loc}(Base, Offset', 0) : T'}{\text{makeTruth}(Offset \neq_{Int} Offset')}$$

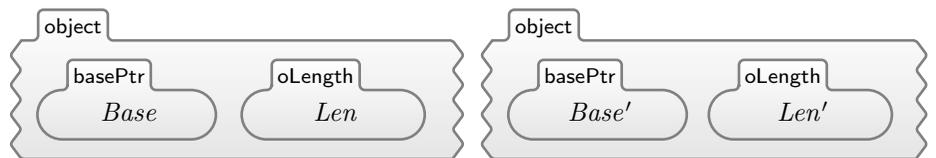
when $\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')$

RULE COMPARE-EQ-DIFFERENT-OBJECTS

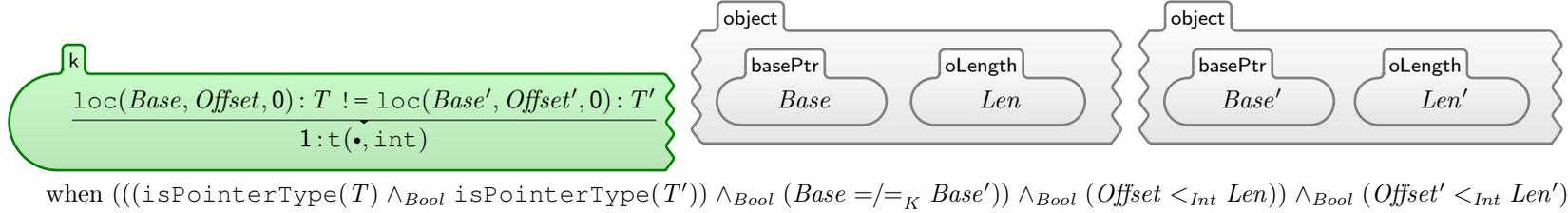
k

$$\frac{\text{loc}(Base, Offset, 0) : T == \text{loc}(Base', Offset', 0) : T'}{0 : t(\cdot, int)}$$

when $((\text{isPointerType}(T) \wedge_{Bool} \text{isPointerType}(T')) \wedge_{Bool} (Base \neq_K Base')) \wedge_{Bool} (Offset <_{Int} Len) \wedge_{Bool} (Offset' <_{Int} Len')$



RULE COMPARE-NEQ-DIFFERENT-OBJECTS



RULE EQUAL-NULL-LEFT

$$\frac{N : T}{\text{NullPointer} : T'} == \text{---} : T'$$

when (isPromoted(T) \wedge_{Bool} isPointerType(T')) \wedge_{Bool} ($N ==_K \text{NullPointerConstant}$)

RULE EQUAL-NULL-RIGHT

$$\text{---} : T == \frac{N : T'}{\text{NullPointer} : T}$$

when (isPointerType(T) \wedge_{Bool} isPromoted(T')) \wedge_{Bool} ($N ==_K \text{NullPointerConstant}$)

RULE NEQUAL-NULL-LEFT

$$\frac{N : T}{\text{NullPointer} : T'} != \text{---} : T'$$

when (isPromoted(T) \wedge_{Bool} isPointerType(T')) \wedge_{Bool} ($N ==_K \text{NullPointerConstant}$)

RULE NEQUAL-NULL-RIGHT

$$\text{---} : T != \frac{N : T'}{\text{NullPointer} : T}$$

when (isPointerType(T) \wedge_{Bool} isPromoted(T')) \wedge_{Bool} ($N ==_K \text{NullPointerConstant}$)

END MODULE

MODULE DYNAMIC-SEMANTICS-BITWISE

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.10 ¶3–4 The usual arithmetic conversions are performed on the operands.

The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

RULE

$$\frac{I_1 : T \ \& \ I_2 : T}{\text{arithInterpret}(T, I_1 \ \&_{Int} \ I_2)}$$

when isPromoted(T)

(n1570) §6.5.11 ¶3–4 The usual arithmetic conversions are performed on the operands.

The result of the \wedge operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

RULE

$$\frac{I_1 : T \ \wedge \ I_2 : T}{\text{arithInterpret}(T, I_1 \ \oplus_{Int} \ I_2)}$$

when isPromoted(T)

(n1570) §6.5.12 ¶3–4 The usual arithmetic conversions are performed on the operands.

The result of the $|$ operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

RULE

$$\frac{I_1 : T \ \mid \ I_2 : T}{\text{arithInterpret}(T, I_1 \ \mid_{Int} \ I_2)}$$

when isPromoted(T)

END MODULE

MODULE DYNAMIC-SEMANTICS-LOGICAL

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

Here, we wrapped the controlling expressions with `simplifyTruth` when heating them, so that we are guaranteed the values in those locations are either `tv(0, int)` or `tv(1, int)`.

(n1570) §6.5.13 ¶3–4 The `&&` operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type `int`.

Unlike the bitwise binary `&` operator, the `&&` operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.

RULE

$$\frac{0:t(-, \text{int}) \ \&\& \ E}{0:t(\bullet, \text{int})}$$

RULE

$$\frac{1:t(-, \text{int}) \ \&\& \ E}{\text{sequencePoint} \curvearrowright \text{simplifyTruth}(E)}$$

RULE

$$\frac{V \ \&\& \ _}{\text{simplifyTruth}(V)}$$

when `isNotTruthValue(V)`

(n1570) §6.5.14 ¶3–4 The `||` operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type `int`.

Unlike the bitwise `|` operator, the `||` operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated.

RULE

$$\frac{0:t(-, \text{int}) \ || \ E}{\text{sequencePoint} \curvearrowright \text{simplifyTruth}(E)}$$

RULE

$$\frac{1:t(-, \text{int}) \mid \mid E}{1:t(\bullet, \text{int})}$$

RULE

$$\frac{V \mid \mid -}{V \neq 0:t(\bullet, \text{int})}$$

when isNotTruthValue(V)

END MODULE

MODULE DYNAMIC-SEMANTICS-CONDITIONAL-EXPRESSION

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

SYNTAX $K ::= \text{getTypes}(\text{List}\{K\})$
 $\mid \text{types}(\text{List}\{K\})$

CONTEXT: $\text{types}(-, \square, -)$

SYNTAX $K ::= \text{convertedType}(K)$ [strict]

RULE

$$\frac{\text{getTypes}(L)}{\text{types}(\text{wrapWithTypeOf}(L))}$$

SYNTAX $\text{List}\{K\} ::= \text{wrapWithTypeOf}(\text{List}\{K\})$ [function]

DEFINE

$$\frac{\text{wrapWithTypeOf}(K, L)}{\text{retype}(\text{typeof}(K), \text{wrapWithTypeOf}(L))}$$

DEFINE

$\frac{\text{wrapWithTypeOf}(\bullet)}{\bullet}$

SYNTAX $K ::= \text{retype}(K)$ [function strict]

DEFINE

$\frac{\text{retype}(T)}{\text{t}(\bullet, \text{pointerType}(\text{innerType}(T)))}$
when isArrayType(T)

DEFINE

$\frac{\text{retype}(T)}{\text{t}(\bullet, \text{pointerType}(T))}$
when isFunctionType(T)

DEFINE

$\frac{\text{retype}(T)}{T}$
when $\neg_{Bool} \left(\bigvee_{Bool} \text{isFunctionType}(T) \right)$

(n1570) §6.5.15 ¶4 The first operand is evaluated; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated). The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.

RULE

$\frac{\bullet}{\text{getTypes}(E_1, E_2)} \rightsquigarrow (E ? E_1 : E_2)$

(n1570) §6.5.15 ¶5 If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.

RULE

$$\frac{\text{types}(T_1, T_2) \curvearrowright (E ? E_1 : E_2)}{\text{convertedType}(\text{usualArithmeticConversion}(T_1, T_2))}$$

when $((T_1 \neq_K T_2) \wedge_{Bool} \text{isArithmeticType}(T_1)) \wedge_{Bool} \text{isArithmeticType}(T_2)$

(n1570) §6.5.15 ¶6 If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one operand is a pointer to void or a qualified version of **void**, in which case the result type is a pointer to an appropriately qualified version of **void**.

RULE **CONDITIONAL-LEFT-IS-NULL**

$$\frac{\text{types}(T_1, T_2) \curvearrowright (E ? 0 : T_1 : E_2)}{\text{convertedType}(T_2)}$$

when $\text{hasIntegerType}(T_1) \wedge_{Bool} \text{isPointerType}(T_2)$

RULE **CONDITIONAL-RIGHT-IS-NULL**

$$\frac{\text{types}(T_1, T_2) \curvearrowright (E ? E_1 : 0 : T_2)}{\text{convertedType}(T_1)}$$

when $\text{hasIntegerType}(T_2) \wedge_{Bool} \text{isPointerType}(T_1)$

RULE

$$\frac{\text{types}(T_1, T_2) \curvearrowright (E ? E_1 : E_2)}{\text{convertedType}(T_1)}$$

when $\text{isPointerType}(T_1) \wedge_{Bool} \text{isPointerType}(T_2)$

RULE

$$\frac{\text{types}(T, T) \rightsquigarrow (E ? E_1 : E_2)}{\text{convertedType}(T)}$$

when $\neg_{Bool} \text{isPointerType}(T)$

RULE

$$\frac{\text{convertedType}(T) \rightsquigarrow (E ? E_1 : E_2)}{\text{IfThenElse}(E, \text{cast}(T, E_1), \text{cast}(T, E_2))}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-ASSIGNMENT

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.16 ¶3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

(n1570) §6.5.16.1 ¶2 In simple assignment (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

RULE ASSIGN

$$\frac{\text{lv}(Loc, T) := V : T}{\text{write}(\text{lv}(Loc, T), V : T) \rightsquigarrow V : T}$$

RULE CONVERT-FOR-ASSIGNMENT

$$\text{lval}(_, T) := \frac{V : T'}{\text{cast}(T, V : T')}$$

when $T \neq_K T'$

(n1570) §6.5.16.1 ¶3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

(n1570) §6.5.16.2 ¶3 A compound assignment of the form $E_1 \text{ op} = E_2$ is equivalent to the simple assignment expression $E_1 = E_1 \text{ op} (E_2)$, except that the lvalue E_1 is evaluated only once, and with respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. If E_1 has an atomic type, compound assignment is a read-modify-write operation with `memory_order_seq_cst` memory order semantics.

SYNTAX $K ::= \text{compoundAssignment}(KLabel, K, K)$

CONTEXT: $\text{compoundAssignment}(_, \frac{\square}{\text{peval}(\square)}, _)$

CONTEXT: $\text{compoundAssignment}(_, _, \frac{\square}{\text{reval}(\square)})$

RULE COMPOUNDASSIGNMENT-MULT

$$\frac{E_1 * = E_2}{\text{compoundAssignment}(_ * _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-DIV

$$\frac{E_1 /= E_2}{\text{compoundAssignment}(_ / _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-MODULO

$$\frac{k \quad E_1 \% = E_2}{\text{compoundAssignment}(_ \% _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-PLUS

$$\frac{k \quad E_1 += E_2}{\text{compoundAssignment}(_ + _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-MINUS

$$\frac{k \quad E_1 -= E_2}{\text{compoundAssignment}(_ - _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-LEFT-SHIFT

$$\frac{k \quad E_1 \gg = E_2}{\text{compoundAssignment}(_ \ll _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-RIGHT-SHIFT

$$\frac{k \quad E_1 \ll = E_2}{\text{compoundAssignment}(_ \gg _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-BITWISE-AND

$$\frac{k \quad E_1 \& = E_2}{\text{compoundAssignment}(_ \& _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-BITWISE-XOR

$$\frac{k \quad E_1 \wedge = E_2}{\text{compoundAssignment}(_ \wedge _, E_1, E_2)}$$

RULE COMPOUNDASSIGNMENT-BITWISE-OR

$$\frac{E_1 \mid= E_2}{\text{compoundAssignment}(_ \mid _, E_1, E_2)}$$

RULE

$$\frac{\text{compoundAssignment}(L, V, V')}{V := (L(\text{reval}(V), V'))}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-SEQUENCING

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

(n1570) §6.5.17 ¶2 The left operand of a comma operator is evaluated as a void expression; there is a sequence point between its evaluation and that of the right operand. Then the right operand is evaluated; the result has its type and value.

RULE

$$\frac{\text{Comma}(\text{List}(V, K', L))}{\text{sequencePoint} \curvearrowright \text{Comma}(\text{List}(K', L))}$$

RULE

$$\frac{\text{Comma}(\text{List}(K))}{K}$$

END MODULE

MODULE DYNAMIC-C-EXPRESSIONS

IMPORTS DYNAMIC-SEMANTICS-EXPRESSIONS-INCLUDE

IMPORTS DYNAMIC-SEMANTICS-COMPOUND-LITERAL
IMPORTS DYNAMIC-SEMANTICS-LOGICAL
IMPORTS DYNAMIC-SEMANTICS-CONDITIONAL-EXPRESSION
IMPORTS DYNAMIC-SEMANTICS-SIZEOF
IMPORTS DYNAMIC-SEMANTICS-IDENTIFIERS
IMPORTS DYNAMIC-SEMANTICS-FUNCTION-CALLS
IMPORTS DYNAMIC-SEMANTICS-ARRAY-SUBSCRIPTING
IMPORTS DYNAMIC-SEMANTICS-CAST
IMPORTS DYNAMIC-SEMANTICS-ASSIGNMENT
IMPORTS DYNAMIC-SEMANTICS-LITERALS
IMPORTS DYNAMIC-SEMANTICS-BITWISE
IMPORTS DYNAMIC-SEMANTICS-BITWISE-SHIFT
IMPORTS DYNAMIC-SEMANTICS-MULTIPLICATIVE-OPERATORS
IMPORTS DYNAMIC-SEMANTICS-ADDITIVE-OPERATORS
IMPORTS DYNAMIC-SEMANTICS-RELATIONAL
IMPORTS DYNAMIC-SEMANTICS-EQUALITY
IMPORTS DYNAMIC-SEMANTICS-UNARY-ARITHMETIC
IMPORTS DYNAMIC-SEMANTICS-MEMBERS
IMPORTS DYNAMIC-SEMANTICS-DEREFERENCE
IMPORTS DYNAMIC-SEMANTICS-REFERENCE

```
IMPORTS DYNAMIC-SEMANTICS-POSTFIX-INCREMENT-AND-DECREMENT
```

```
IMPORTS DYNAMIC-SEMANTICS-PREFIX-INCREMENT-AND-DECREMENT
```

```
IMPORTS DYNAMIC-SEMANTICS-SEQUENCING
```

```
END MODULE
```

A.4 Statements

This section represents the semantics of C statements, and generally corresponds to §6.8 in the C standard. The first part of this section gives a mechanism for calculating `goto`-maps. This is a static pass that saves all relevant information necessary to execute `gotos` to particular labels. The second part of this section focuses on the dynamic semantics of statements.

The static parts of this and of Section A.6 are implementation-like, and have a very strong algorithmic quality. Although we describe them semantically here, they could also have been implemented in a parser or front-end like CIL [114].

MODULE COMMON-SEMANTICS-STATEMENTS-INCLUDE

IMPORTS COMMON-INCLUDE

SYNTAX *Statement* ::= loopMarked

SYNTAX *K* ::= genLabel(*Nat*, *K*)
 | popLoop
 | popBlock
 | frozenDeclaration(*Nat*, *Nat*, *K*)
 | gotoObj(*Nat*, *List*, *K*, *List*, *List*)
 | case(*Nat*)

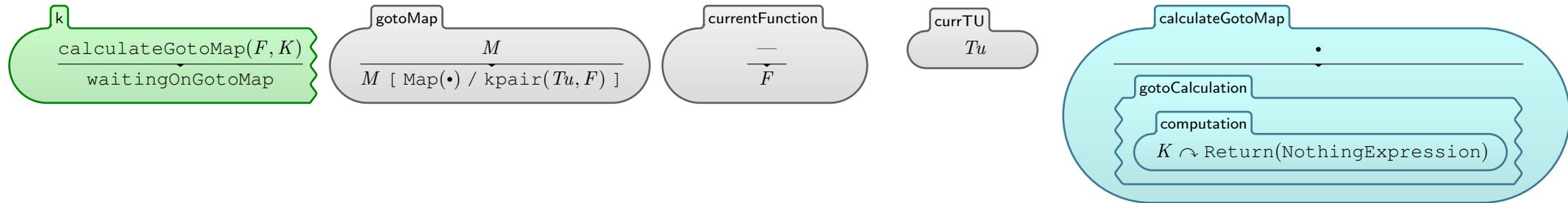
END MODULE

MODULE COMMON-SEMANTICS-PROCESS-LABELS

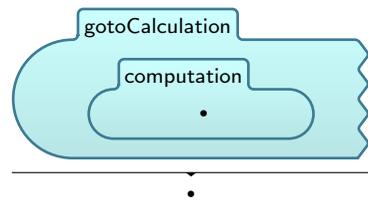
IMPORTS COMMON-SEMANTICS-STATEMENTS-INCLUDE

SYNTAX *K* ::= waitingOnGotoMap

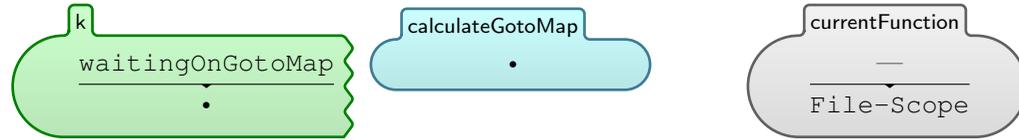
RULE



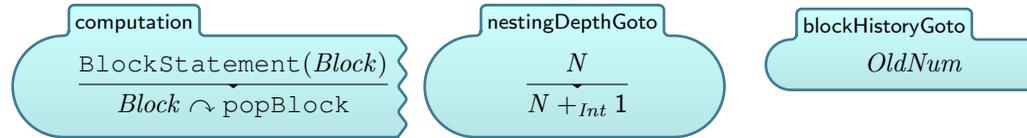
RULE



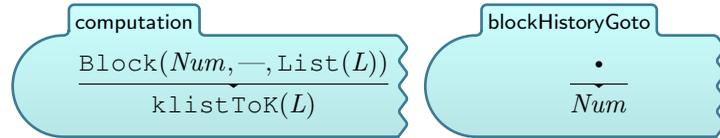
RULE

SYNTAX $K ::= \text{endBlockForGoto}(Nat)$

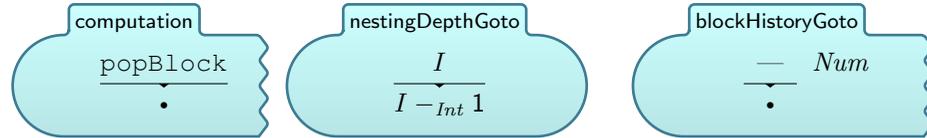
RULE



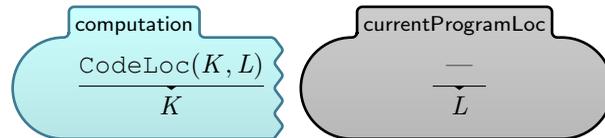
RULE



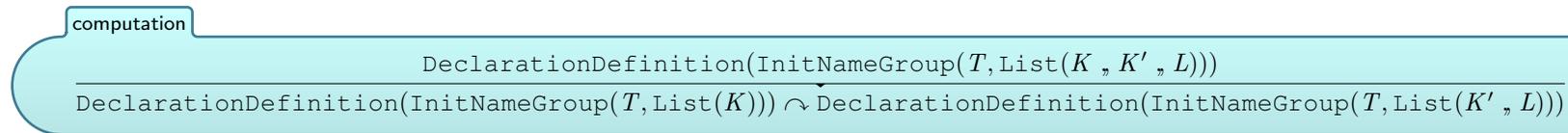
RULE

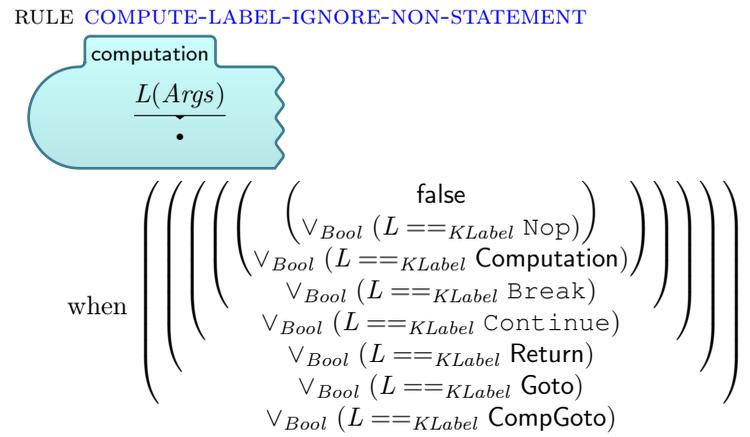
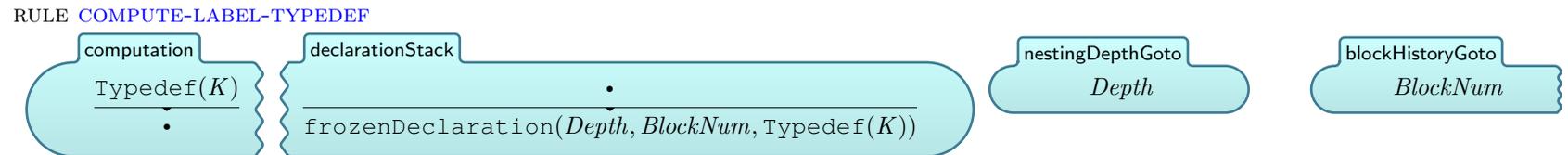
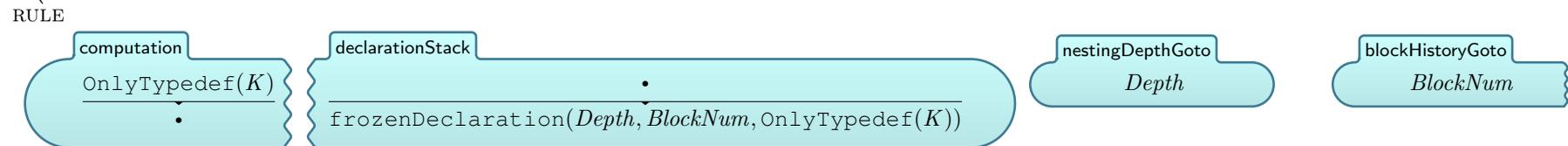
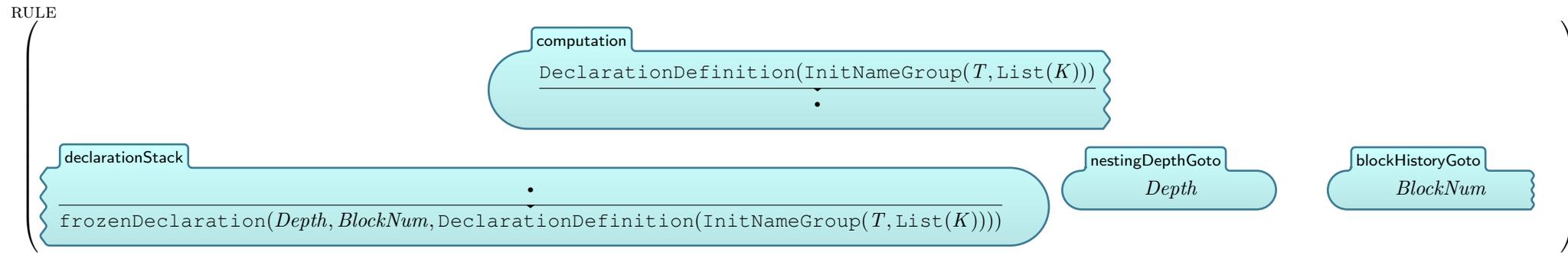
when $I >_{Int} 0$

RULE DEFINITIONLOC-COMPUTATION

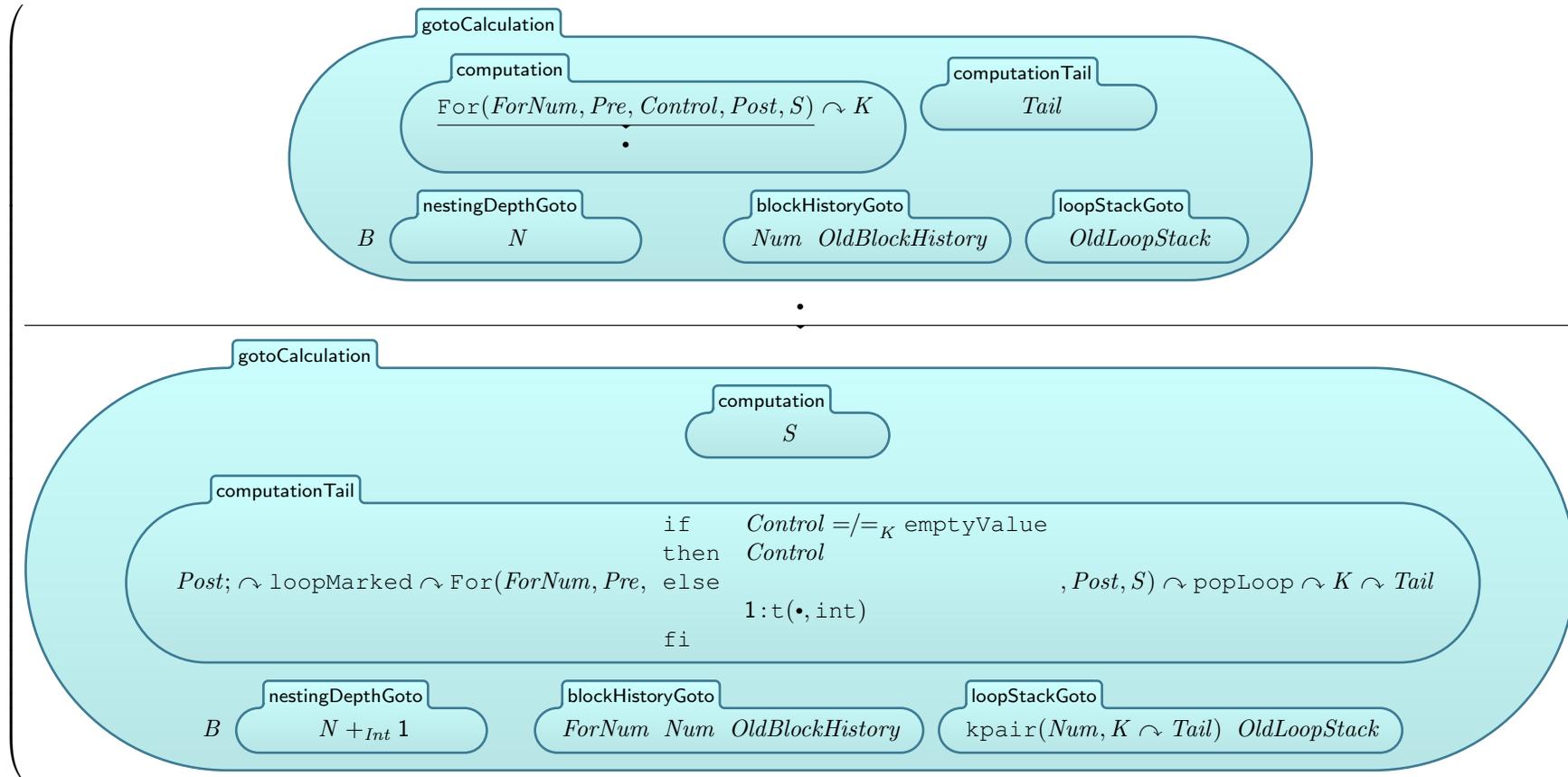


RULE

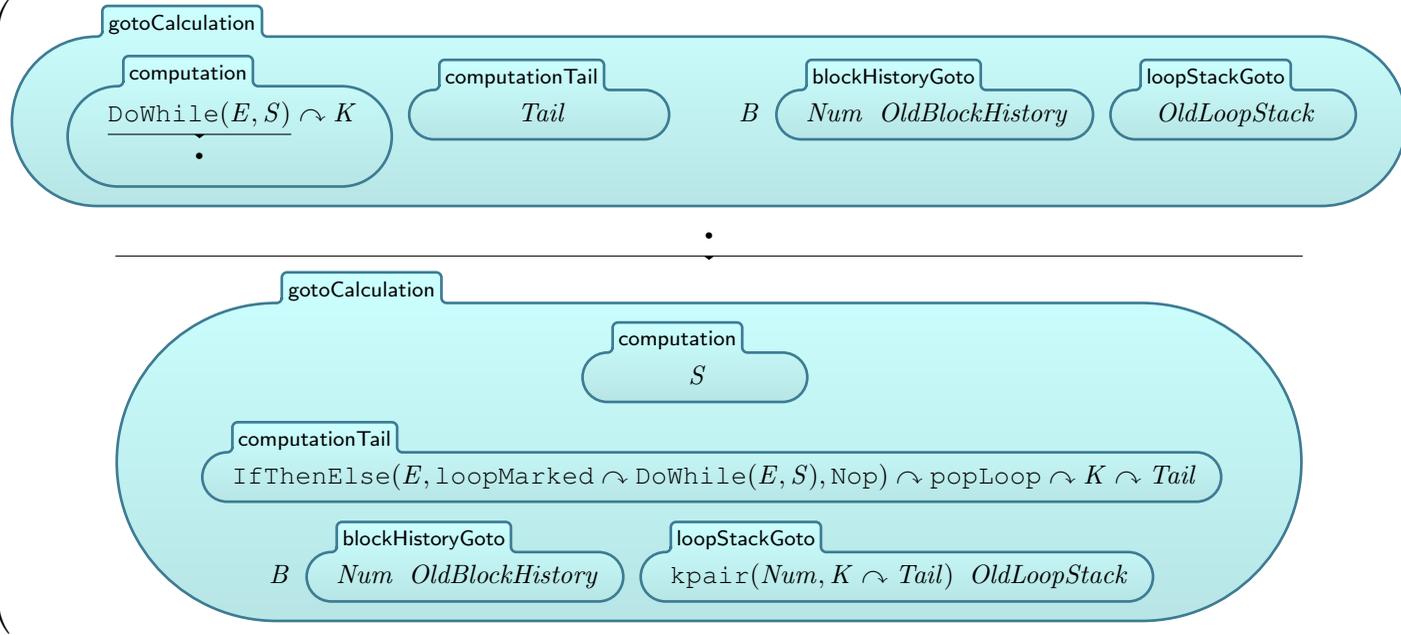




RULE COMPUTE-LABEL-FOR



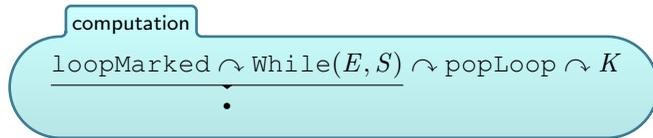
RULE COMPUTE-LABEL-DO-WHILE



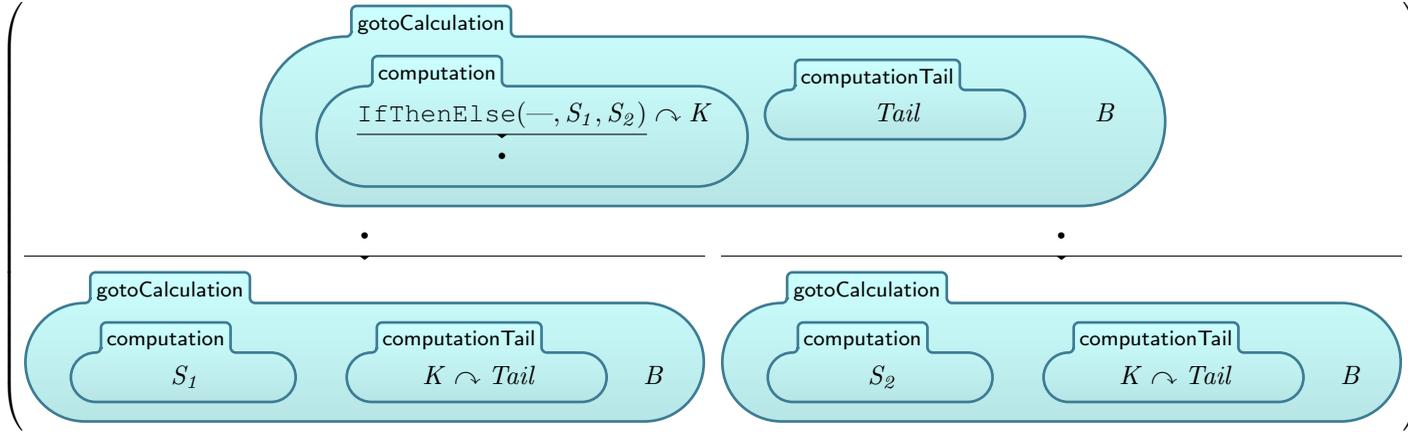
RULE COMPUTE-LABEL-WHILE-MARK



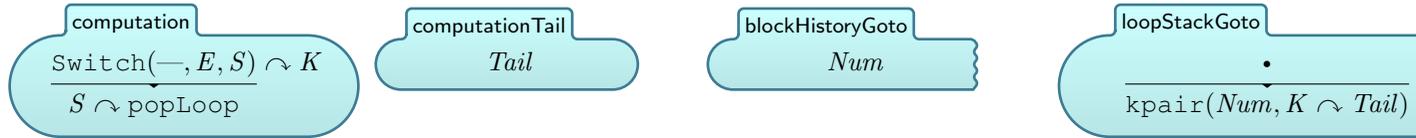
RULE COMPUTE-LABEL-WHILE-DONE



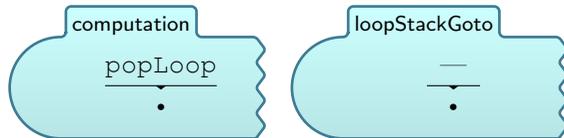
RULE COMPUTE-LABEL-IF-THEN-ELSE



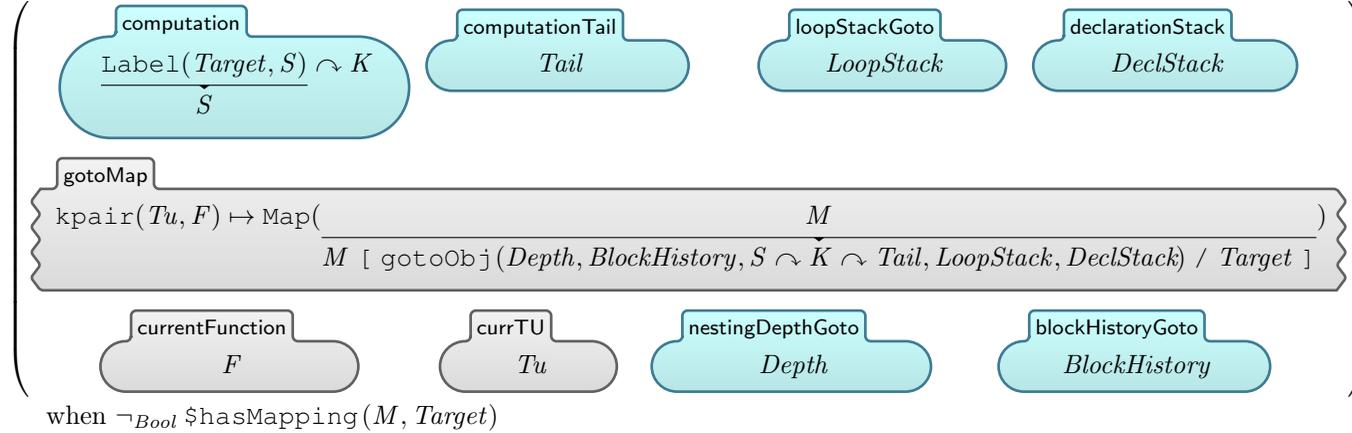
RULE COMPUTE-LABEL-SWITCH



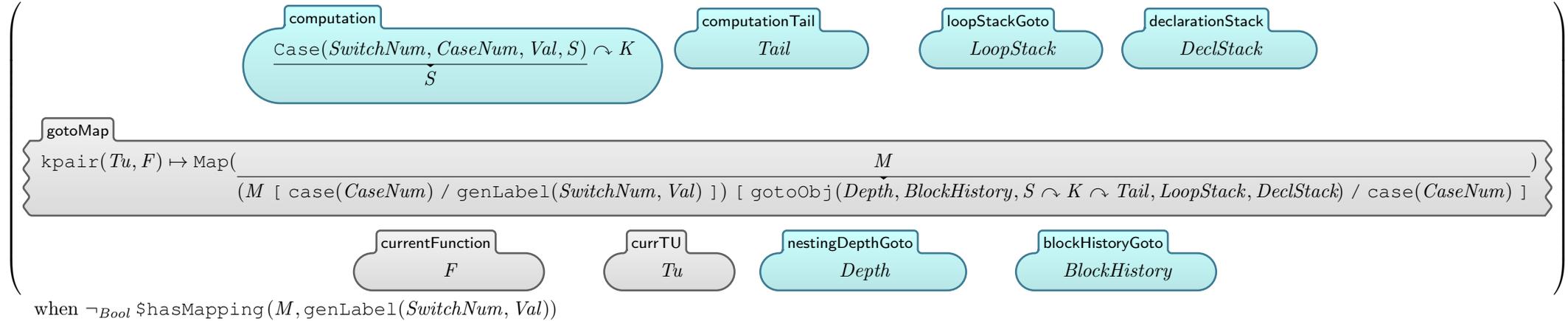
RULE COMPUTE-LABEL-POPLOOP

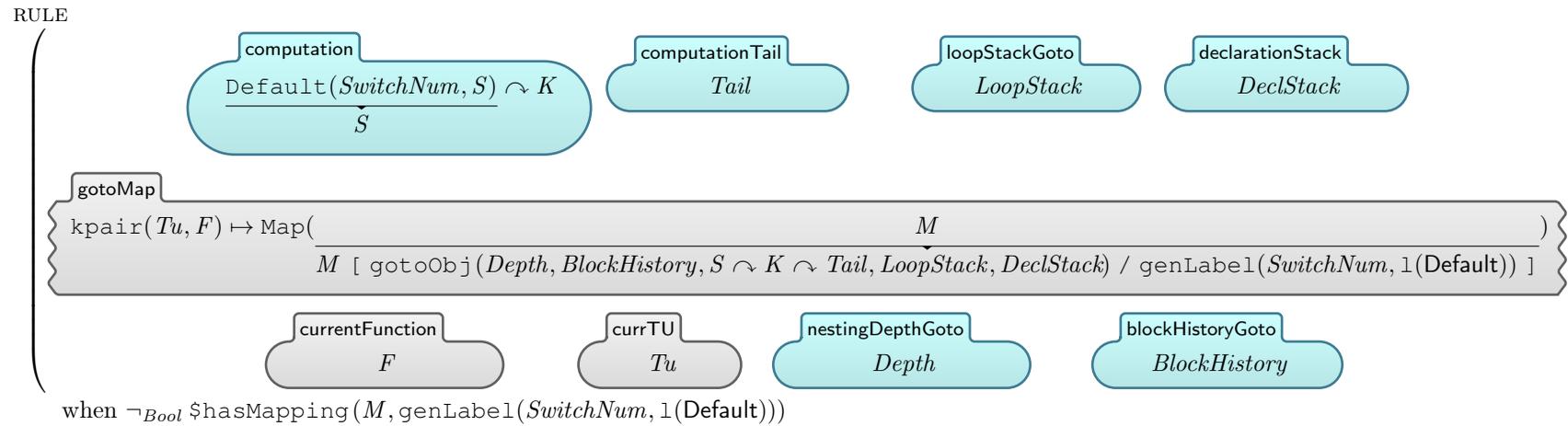


RULE



RULE





END MODULE

MODULE COMMON-C-STATEMENTS

IMPORTS COMMON-SEMANTICS-STATEMENTS-INCLUDE

IMPORTS COMMON-SEMANTICS-PROCESS-LABELS

END MODULE

MODULE DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

IMPORTS COMMON-SEMANTICS-STATEMENTS-INCLUDE

IMPORTS DYNAMIC-INCLUDE

SYNTAX $K ::= \text{pushBlock}$
 $\quad \quad \quad | \text{addToHist}(Nat)$

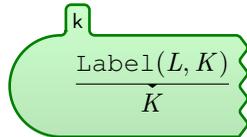
END MODULE

MODULE DYNAMIC-SEMANTICS-LABELED-STATEMENTS

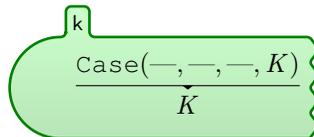
IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.1 ¶4 Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

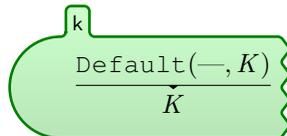
RULE SKIP-LABEL



RULE CASE-FALL-THROUGH



RULE DEFAULT-FALL-THROUGH



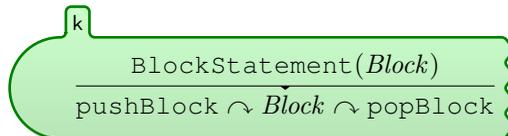
END MODULE

MODULE DYNAMIC-SEMANTICS-BLOCKS

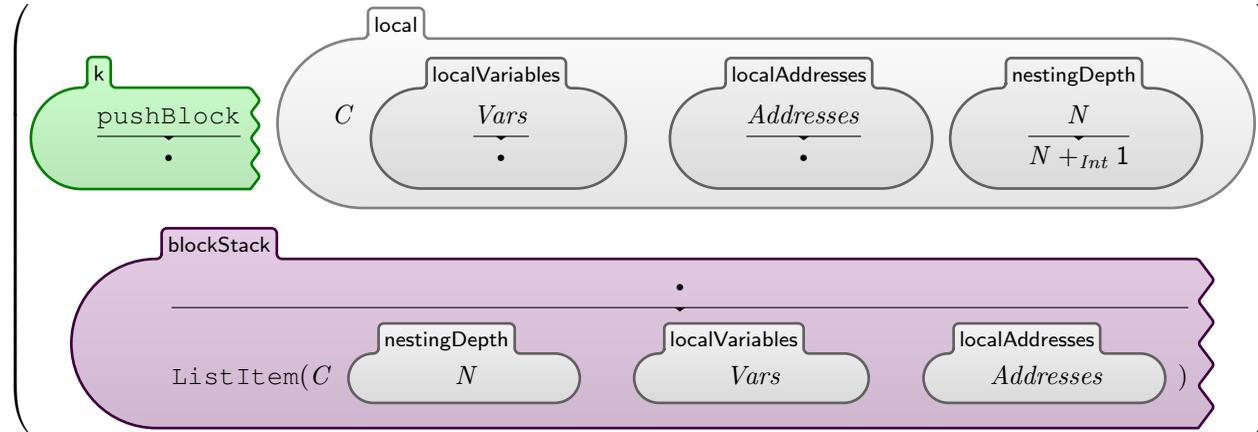
IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.2 ¶2 A *compound statement* is a block.

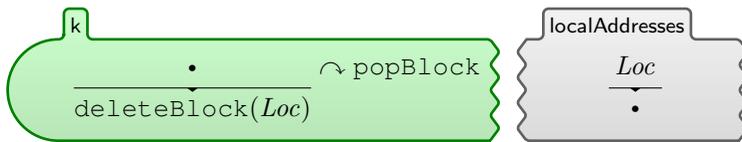
RULE



RULE PUSH-BLOCK



RULE POP-BLOCK-FREE-MEMORY

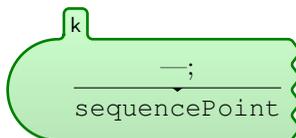


RULE POP-BLOCK



(n1570) §6.8.3 ¶2 The expression in an expression statement is evaluated as a void expression for its side effects.

RULE EXPRESSION-STATEMENT



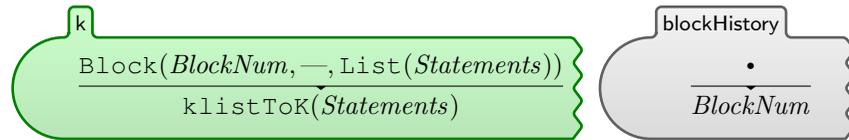
(n1570) §6.8.3 ¶3 A null statement (consisting of just a semicolon) performs no operations.

RULE

$$\frac{\text{Nop}}{\cdot}$$

(n1570) §6.8 ¶3 A block allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, and within each declaration in the order that declarators appear.

RULE DISSOLVE-BLOCK



END MODULE

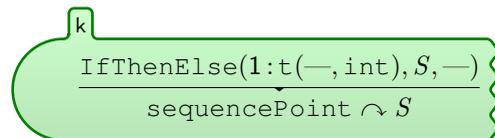
MODULE DYNAMIC-SEMANTICS-IF-THEN

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

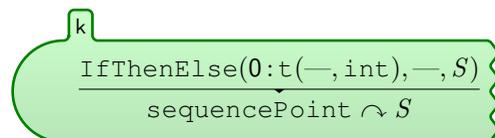
(n1570) §6.8 ¶4 ... There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

(n1570) §6.8.4.1 ¶10 In both forms, the first substatement is executed if the expression compares unequal to 0. In the **else** form, the second substatement is executed if the expression compares equal to 0. If the first substatement is reached via a label, the second substatement is not executed.

RULE IF-THEN-ELSE-TRUE



RULE IF-THEN-ELSE-FALSE



RULE

$$\frac{\text{IfThenElse}(\frac{V}{\text{simplifyTruth}(V)}, -, -)}{\text{when isNotTruthValue}(V)}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-SWITCH

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

SYNTAX $K ::= \text{handleSwitch}(Nat, K) \text{ [strict(2)]}$
 $| \text{handleSwitch-aux}(K, Value, K)$

(n1570) §6.8.4.2 ¶4 A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. A **case** or **default** label is accessible only within the closest enclosing **switch** statement.

(n1570) §6.8.4.2 ¶5 The integer promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** label. Otherwise, if there is a default label, control jumps to the labeled statement. If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed.

RULE

$$\frac{\text{Switch}(SN, V : T, -)}{\text{sequencePoint} \curvearrow \text{handleSwitch}(SN, \text{cast}(\text{promote}(T), V : T))}$$

when hasIntegerType(T)

RULE

$$\frac{\text{handleSwitch}(SN, V)}{\text{handleSwitch-aux}(SN, V, \text{Map}(M))}$$

currentFunction F currTU Tu gotoMap $\text{kpair}(Tu, F) \mapsto \text{Map}(M)$

SYNTAX $K ::= \text{tryCase}(K, Value, K)$

CONTEXT: $\text{tryCase}(\frac{\square}{\text{reval}(\square)}, -, -)$

RULE

k

$$\frac{\bullet}{\text{tryCase}(Exp, V, CaseHelper)} \curvearrowright \text{handleSwitch-aux}(SN, V, \text{Map}(- \text{genLabel}(SN, Exp) \mapsto CaseHelper))$$

when $Exp \neq_K \perp(\text{Default})$

RULE

k

$$\frac{\text{handleSwitch-aux}(SN, -, \text{Map}(\text{genLabel}(SN, \perp(\text{Default})) \mapsto -))}{\text{Goto}(\text{genLabel}(SN, \perp(\text{Default})))}$$

RULE

k

$$\frac{\text{handleSwitch-aux}(-, -, \text{Map}(\bullet))}{\bullet}$$

RULE

k

$$\text{handleSwitch-aux}(SN, -, \text{Map}(- \text{genLabel}(SN', -) \mapsto -))$$

when $SN \neq_{Int} SN'$

RULE

k

$$\text{handleSwitch-aux}(SN, -, \text{Map}(- (L(-)) \mapsto -))$$

when $L \neq_{KLabel} \text{genLabel}$

RULE

$$\frac{\text{tryCase}(\frac{V : T'}{\text{cast}(T, V : T')}, \text{---} : T, \text{---})}{\text{when } T \neq_K T'}$$

RULE

$$\frac{\text{tryCase}(V' : T, V : T, \text{CaseHelper})}{\text{when } V \neq_K V'}$$

RULE

$$\frac{\text{tryCase}(V, V, \text{CaseHelper})}{\text{Goto}(\text{CaseHelper})}$$

END MODULE

MODULE DYNAMIC-SEMANTICS-WHILE

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.5.1 ¶1 The evaluation of the controlling expression takes place before each execution of the loop body.

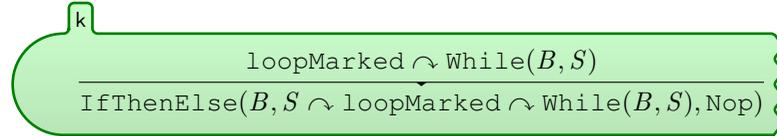
RULE WHILE-MARK

$$\frac{\text{While}(B, S) \rightsquigarrow K}{\text{loopMarked} \rightsquigarrow \text{While}(B, S) \rightsquigarrow \text{popLoop}}$$

$$\frac{\text{blockHistory}}{\text{Num}}$$

$$\frac{\text{loopStack}}{\text{kpair}(\text{Num}, K)}$$

RULE WHILE



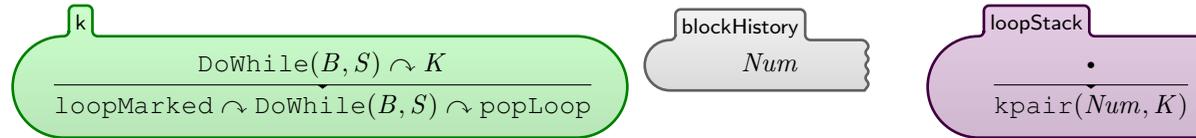
END MODULE

MODULE DYNAMIC-SEMANTICS-DO-WHILE

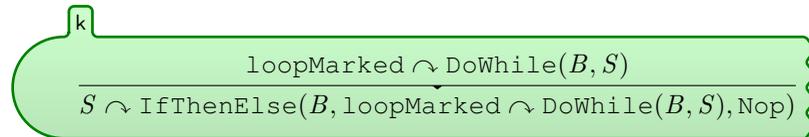
IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.5.2 ¶1 The evaluation of the controlling expression takes place after each execution of the loop body.

RULE DO-WHILE-MARK



RULE DO-WHILE



END MODULE

MODULE DYNAMIC-SEMANTICS-FOR

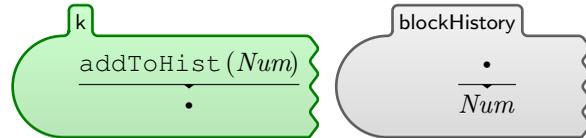
IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.5.2 ¶1 The statement **for** (*clause-1*; *expression-2*; *expression-3*) *statement* behaves as follows: The expression *expression-2* is the controlling expression that is evaluated before each execution of the loop body. The expression *expression-3* is evaluated as a void expression after each execution of the loop body. If *clause-1* is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.

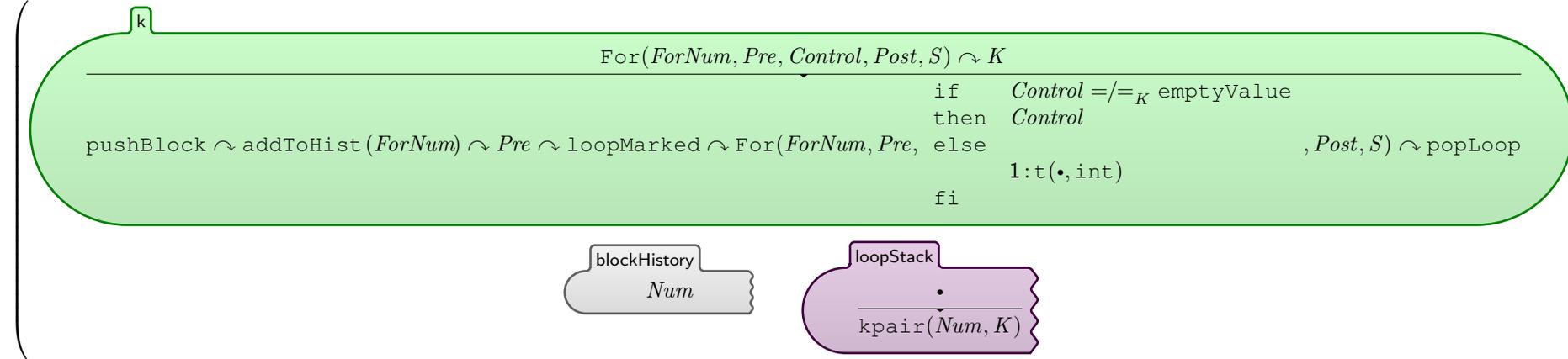
(n1570) §6.8.5.2 ¶1 Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2* is replaced by a nonzero constant.

RULE
 $\frac{\text{ForClauseExpression}(K)}{K;}$

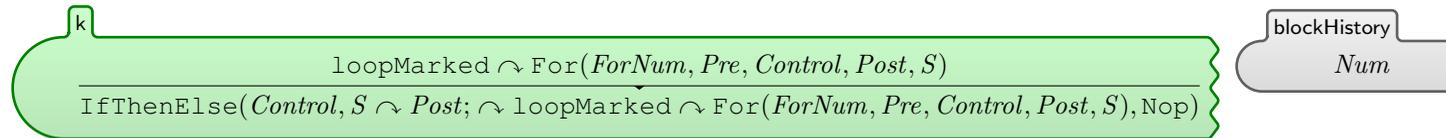
RULE



RULE FOR-MARK



RULE FOR



END MODULE

MODULE DYNAMIC-SEMANTICS-GOTO

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.6.1 ¶2 A **goto** statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

(n1570) §6.2.4 ¶6 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

SYNTAX $K ::= \text{processGoto}(K)$
 $\quad \quad \quad | \text{processGotoDown}(K)$
 $\quad \quad \quad | \text{processGotoSameBlock}(List, List)$

RULE



RULE



when $(L ==_{KLabel} \text{processGoto})$
 $\vee_{Bool} (L ==_{KLabel} \text{processGotoDown})$

RULE PROCESSGOTO-POP-DIFFERING-1



when $\neg_{Bool} (BlockNum \text{ in } TargetBlockHistory)$

RULE PROCESSGOTO-POP-DIFFERING-2



when $TargetBlockHistory \neq_{List} ActualHistory$

RULE

$$\frac{\text{processGoto}(\text{gotoObj}(TargetDepth, Prefix \ TargetBlock \ ActualHistory, K, LoopStack, DeclStack))}{\text{addVarsForBlock}(CurrentBlock, DeclStack) \curvearrow \text{processGotoDown}(\text{gotoObj}(TargetDepth, Prefix \ CurrentBlock \ ActualHistory, K, LoopStack, DeclStack))}$$

nestingDepth

blockHistory

 $ActualDepth$ $CurrentBlock \ ActualHistory$ when $(ActualDepth +_{Int} 1) \neq_{Int} TargetDepth$

RULE

$$\frac{\cdot}{\text{pushBlock} \curvearrow \text{addToHist}(TargetBlk) \curvearrow \text{addVarsForBlock}(TargetBlk, DeclStk)} \curvearrow \text{processGotoDown}(\text{gotoObj}(TargetDepth, \text{---} \ TargetBlk \ ActualHist, \text{---}, \text{---}, DeclStk))$$

nestingDepth

blockHistory

 $ActualDepth$ $ActualHist$ when $(ActualDepth +_{Int} 1) \neq_{Int} TargetDepth$

RULE

$$\frac{(L(\text{gotoObj}(sNatDepth, TargetBlock \ BlockHistory, K, LoopStack, DeclStack))) \curvearrow \text{---}}{\text{pushBlock} \curvearrow \text{addToHist}(TargetBlock) \curvearrow \text{addVarsForBlock}(TargetBlock, DeclStack) \curvearrow K}$$

nestingDepth

blockHistory

loopStack

 $Depth$ $BlockHistory$ $LoopStack$ when $\left(\begin{array}{l} (L ==_{KLabel} \text{processGoto}) \\ \vee_{Bool} (L ==_{KLabel} \text{processGotoDown}) \end{array} \right) \wedge_{Bool} (sNatDepth ==_{Int} (Depth +_{Int} 1))$ SYNTAX $K ::= \text{addVarsForBlock}(Nat, List)$

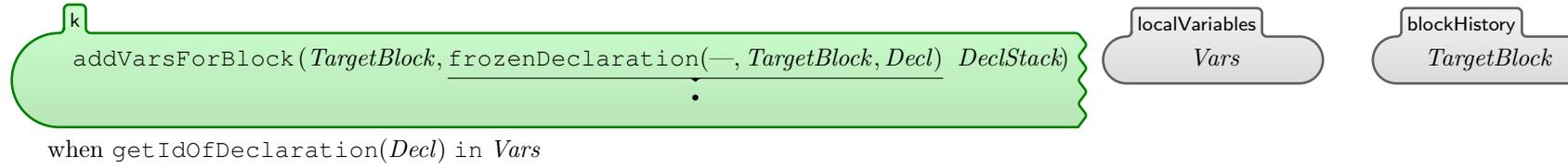
RULE

$$\text{addVarsForBlock}(TargetBlock, \text{frozenDeclaration}(\text{---}, BlockNum, \text{---}) \ DeclStack)$$

blockHistory

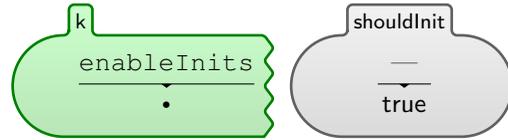
 $TargetBlock$ when $BlockNum \neq_{Int} TargetBlock$

RULE

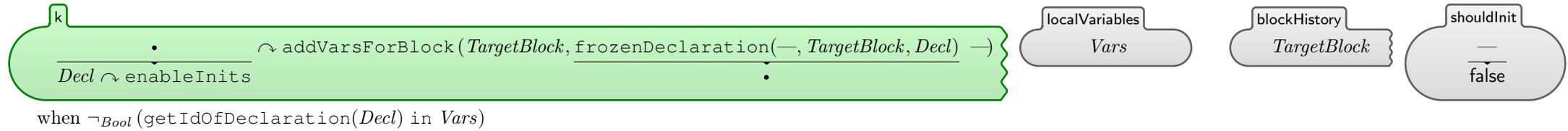


SYNTAX $K ::= \text{enableInits}$

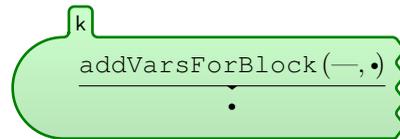
RULE



RULE



RULE



END MODULE

MODULE DYNAMIC-SEMANTICS-CONTINUE

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.6.2 ¶2 A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

<pre> while (...) { ... continue; ... contin: ; } </pre>	<pre> do { ... continue; ... contin: ; } while (...); </pre>	<pre> for (...) { ... continue; ... contin: ; } </pre>
--	---	--

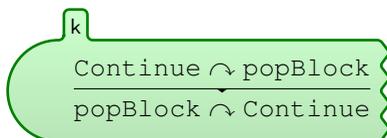
unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto** contin;.

RULE CONTINUE

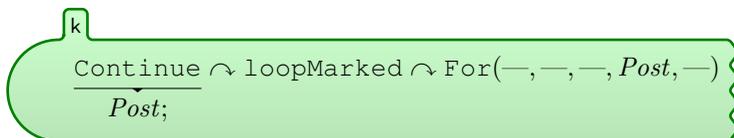


when $((L \neq_{KLabel} \text{loopMarked}) \wedge_{Bool} (L \neq_{KLabel} \text{popBlock})) \wedge_{Bool} (L \neq_{KLabel} \text{pushBlock}) \wedge_{Bool} (L \neq_{KLabel} \text{popLoop})$

RULE CONTINUE-THROUGH-POP



RULE CONTINUE-DONE-FOR



RULE CONTINUE-DONE



when $L \neq_{KLabel} \text{For}$

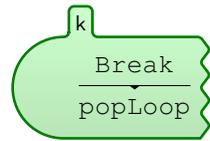
END MODULE

MODULE DYNAMIC-SEMANTICS-BREAK

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

(n1570) §6.8.6.3 ¶2 A break statement terminates execution of the smallest enclosing switch or iteration statement.

RULE BREAK



END MODULE

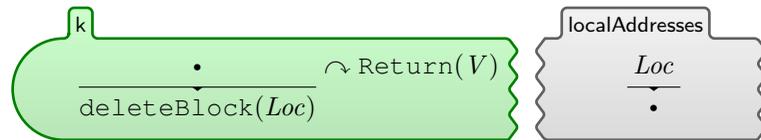
MODULE DYNAMIC-SEMANTICS-RETURN

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

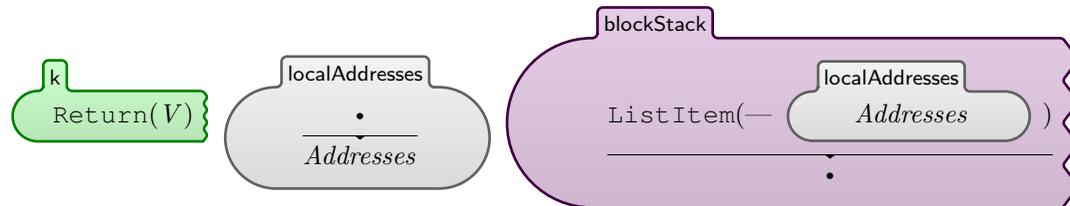
(n1570) §6.8.6.4 ¶2 A return statement terminates execution of the current function and returns control to its caller. A function may have any number of return statements.

(n1570) §6.8.6.4 ¶3 If a return statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.

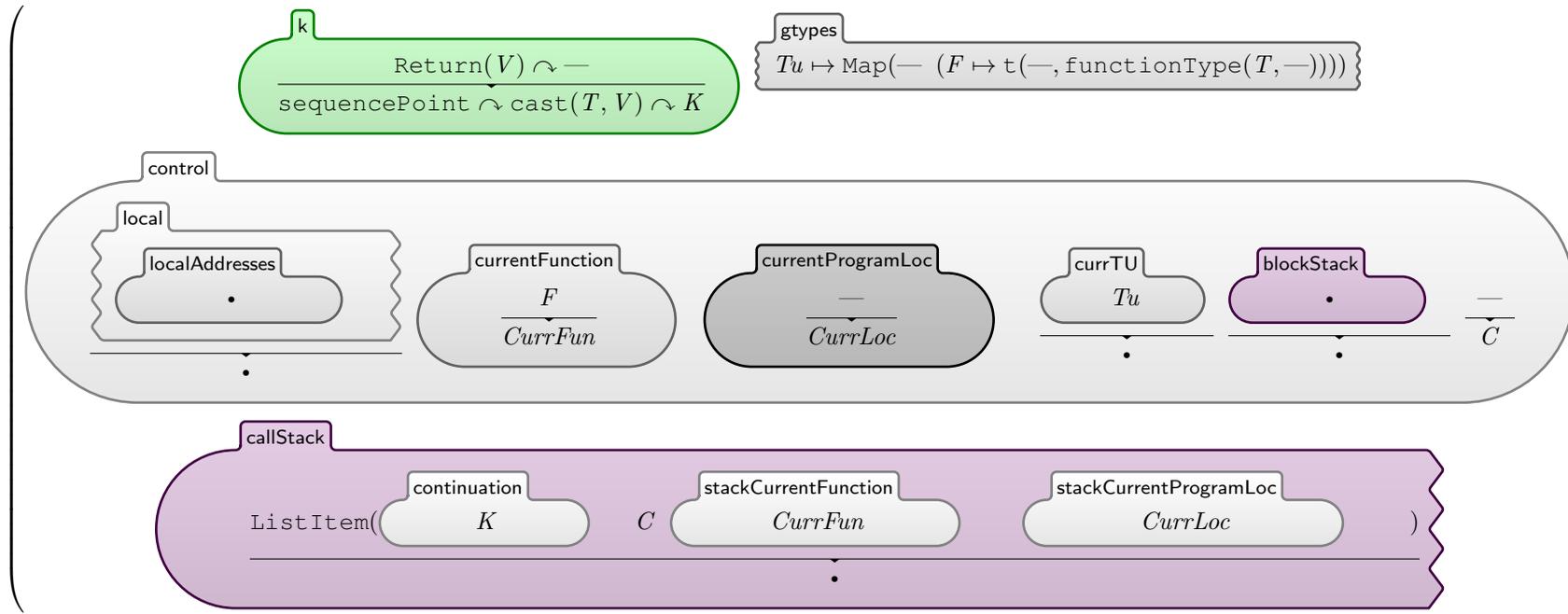
RULE RETURN-CLEAN-LOCAL



RULE FETCH-ALL-LOCALS



RULE RETURN

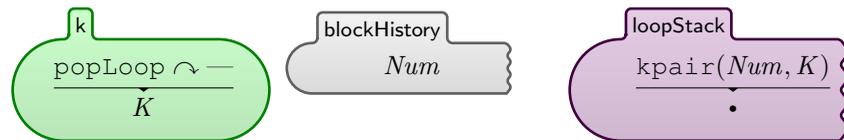


END MODULE

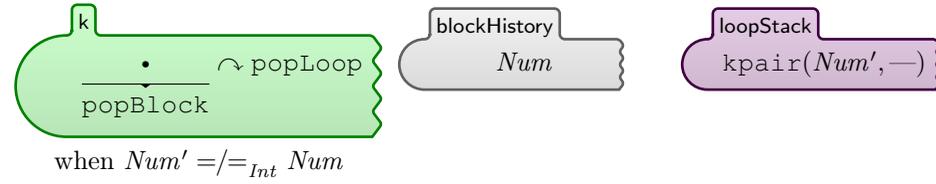
MODULE DYNAMIC-SEMANTICS-STATEMENTS-MISC

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

RULE POPLOOP



RULE POPLOOP-POPBLOCK



END MODULE

MODULE DYNAMIC-C-STATEMENTS

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-INCLUDE

IMPORTS DYNAMIC-SEMANTICS-LABELED-STATEMENTS

IMPORTS DYNAMIC-SEMANTICS-IF-THEN

IMPORTS DYNAMIC-SEMANTICS-FOR

IMPORTS DYNAMIC-SEMANTICS-WHILE

IMPORTS DYNAMIC-SEMANTICS-SWITCH

IMPORTS DYNAMIC-SEMANTICS-GOTO

IMPORTS DYNAMIC-SEMANTICS-RETURN

IMPORTS DYNAMIC-SEMANTICS-BLOCKS

IMPORTS DYNAMIC-SEMANTICS-DO-WHILE

IMPORTS DYNAMIC-SEMANTICS-CONTINUE

IMPORTS DYNAMIC-SEMANTICS-BREAK

IMPORTS DYNAMIC-SEMANTICS-STATEMENTS-MISC

END MODULE

A.5 Typing

This section represents the evaluation of types and the typing of expressions. In our semantics, types are canonicalized (e.g., `long const` is turned into `long-int` with a `const` modifier), which is described in the first part of this section. The typing of expressions (as described in Section 4.2.5) is described in the second part.

MODULE COMMON-SEMANTICS-TYPE-INCLUDE

IMPORTS COMMON-INCLUDE

SYNTAX $K ::= \text{MYHOLE}$
| $\text{addStruct}(Id, List\{KResult\})$
| $\text{addUnion}(Id, List\{KResult\})$
| $\text{canonicalizeType}(Bag)$

SYNTAX $Type ::= \text{extractActualType}(Type)$ [function]

SYNTAX $K ::= \text{evalToType}$

SYNTAX $Set ::= \text{typeStrictLeftBinaryOperators}$ [function]

DEFINE

$\text{typeStrictLeftBinaryOperators}$

$Set(\text{l}(_ \ll _), \text{l}(_ \gg _), \text{l}(_ * _), \text{l}(_ / _), \text{l}(_ \% _), \text{l}(_ + _), \text{l}(_ - _), \text{l}(_ \gg = _), \text{l}(_ \ll = _), \text{l}(_ \& _), \text{l}(_ \wedge _), \text{l}(_ | _), \text{l}(_ ++), \text{l}(_ --), \text{l}(_ -- _), \text{l}(_ ++ _))$

END MODULE

MODULE COMMON-SEMANTICS-TYPE-DECLARATIONS

IMPORTS COMMON-SEMANTICS-TYPE-INCLUDE

SYNTAX $K ::= \text{giveGlobalType}(K, Type)$
| $\text{giveLocalType}(K, Type)$

RULE IGNORE-VOLATILE

$\frac{\text{t}(S, \text{qualifiedType}(\text{t}(S', T), \text{Volatile}))}{\text{t}(S \ S', T)}$

[anywhere]

RULE IGNORE-ATOMIC

$\frac{\text{t}(S, \text{qualifiedType}(\text{t}(S', T), \text{Atomic}))}{\text{t}(S \ S', T)}$

[anywhere]

RULE IGNORE-RESTRICT

$$\frac{\text{t}(S, \text{qualifiedType}(\text{t}(S', T), \text{Restrict}))}{\text{t}(S \ S', T)}$$

[anywhere]

RULE IGNORE-AUTO

$$\frac{\text{t}(S, \text{qualifiedType}(\text{t}(S', T), \text{Auto}))}{\text{t}(S \ S', T)}$$

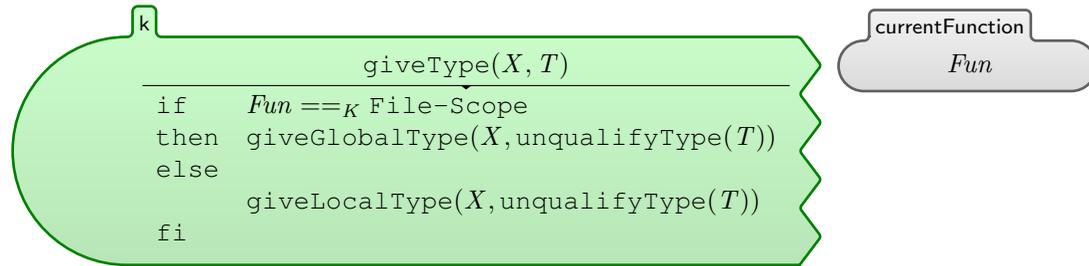
[anywhere]

RULE IGNORE-REGISTER

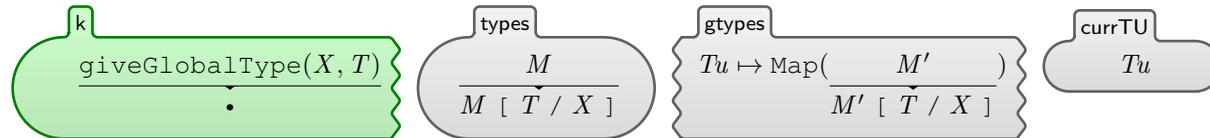
$$\frac{\text{t}(S, \text{qualifiedType}(\text{t}(S', T), \text{Register}))}{\text{t}(S \ S', T)}$$

[anywhere]

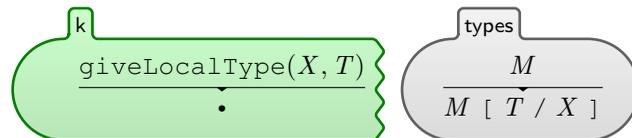
RULE



RULE



RULE



END MODULE

MODULE COMMON-SEMANTICS-TYPE-CANONICALIZATION

IMPORTS COMMON-SEMANTICS-TYPE-INCLUDE

SYNTAX $K ::= \text{canonicalizeType-aux}(Bag, K, Bag, Bag, Bag)$

CONTEXT: $\text{canonicalizeType-aux}(_, \square, _, _, _)$
when $\square =/=_K \bullet$

RULE

$$\frac{\text{canonicalizeType}(B)}{\text{canonicalizeType-aux}(B, \bullet, \bullet, \bullet, \bullet)}$$

RULE

$$\frac{\bullet \curvearrowright \text{canonicalizeType-aux}(_ \text{TAtomic}(K_1, K_2), _, _, _, _)}{\text{DeclType}(K_1, K_2)}$$

RULE

$$\frac{\bullet \curvearrowright \text{canonicalizeType-aux}(_ \text{AlignasType}(K_1, K_2), _, _, _, _)}{\text{DeclType}(K_1, K_2)}$$

RULE

$$\frac{\bullet \curvearrowright \text{canonicalizeType-aux}(_ \text{AlignasExpression}(K_1), _, _, _, _)}{\text{typeof}(K_1)}$$

SYNTAX $K ::= \text{atomic}(Type)$
| $\text{alignas}(Type)$

RULE

$$\frac{T \curvearrowright \text{canonicalizeType-aux}(_ \text{TAtomic}(K_1, K_2), _, _ \frac{\bullet}{\text{atomic}(T)}, _, _)}{\bullet}$$

when $\text{isTypeResult}(T)$

RULE

$$\frac{\text{k} \quad T \rightsquigarrow \text{canonicalizeType-aux}(- \text{AlignasType}(K_1, K_2), -, - \frac{\cdot}{\text{alignas}(T)}, -, -)}{\cdot}$$

when isTypeResult(T)

RULE

$$\frac{\text{k} \quad T \rightsquigarrow \text{canonicalizeType-aux}(- \text{AlignasExpression}(K_1), -, - \frac{\cdot}{\text{alignas}(T)}, -, -)}{\cdot}$$

when isTypeResult(T)

RULE

$$\frac{t(\text{Const } S, \text{arrayType}(t(S', T), N))}{t(S, \text{arrayType}(t(\text{Const } S', T), N))}$$

[anywhere]

RULE

$$\frac{\text{canonicalizeSpecifier}(\text{Named}(X))}{t(\bullet, \text{typedefType}(X, T))}$$

when $X \neq_K \text{Identifier}(\text{""})$

types
typedef(X) $\mapsto T$

RULE

$$\frac{\text{canonicalizeSpecifier}(\text{StructRef}(X))}{t(\bullet, \text{structType}(X))}$$

when $X \neq_K \text{Identifier}(\text{""})$

RULE

$$\frac{\text{canonicalizeSpecifier}(\text{EnumRef}(X))}{t(\bullet, \text{enumType}(X))}$$

when $X \neq_K \text{Identifier}(\text{""})$

RULE

$$\frac{\text{canonicalizeSpecifier}(\text{UnionRef}(X))}{t(\bullet, \text{unionType}(X))}$$

when $X \neq_K \text{Identifier}(\text{""})$

RULE

$$\frac{\text{canonicalizeSpecifier}(\text{EnumDef}(X, L))}{\text{EnumDef}(X, L) \rightsquigarrow t(\bullet, \text{enumType}(X))}$$

when $X \neq_K \text{Identifier}(\text{""})$

RULE

$$\frac{\text{canonicalizeSpecifier}\left(\frac{L(\text{Identifier}(\text{""}), -)}{\text{unnamed}(N)}\right)}{\left(\begin{array}{l} (L ==_{KLabel} \text{StructDef}) \\ \vee_{Bool} (L ==_{KLabel} \text{EnumDef}) \\ \vee_{Bool} (L ==_{KLabel} \text{UnionDef}) \end{array}\right)}$$

when $\left(\begin{array}{l} (L ==_{KLabel} \text{StructDef}) \\ \vee_{Bool} (L ==_{KLabel} \text{EnumDef}) \\ \vee_{Bool} (L ==_{KLabel} \text{UnionDef}) \end{array}\right)$

$$\frac{\text{freshNat } N}{N +_{Int} 1}$$

RULE

$$\frac{\text{canonicalizeSpecifier}(\text{StructDef}(X, L))}{\text{StructDef}(X, L) \rightsquigarrow t(\bullet, \text{structType}(X))}$$

when $X \neq_K \text{Identifier}(\text{""})$

RULE

$$\frac{\text{canonicalizeSpecifier}(\text{UnionDef}(X, L))}{\text{UnionDef}(X, L) \rightsquigarrow t(\bullet, \text{unionType}(X))}$$

when $X \neq_K \text{Identifier}(\text{""})$

RULE

$$\frac{\text{SpecTypedef}}{\bullet}$$

[anywhere]

RULE IGNORE-INLINE

Inline

•

[anywhere]

RULE IGNORE-NORETURN

Noreturn

•

[anywhere]

RULE

Attribute(—, —)

•

[anywhere]

RULE

k

$\frac{\text{canonicalizeType-aux}(\bullet, \frac{\bullet}{\text{canonicalizeSpecifier}(B)}, B, -, -)}{\bullet}$

when $B \neq_{Bag} \bullet$

RULE

k

$\frac{\text{canonicalizeType-aux}(\bullet, T, \bullet, \bullet, \bullet)}{T}$

SYNTAX $K ::= \text{canonicalizeSpecifier}(Bag)$ [function]

DEFINE

canonicalizeSpecifier(Void)

t(•, void)

DEFINE

canonicalizeSpecifier(Bool)

t(•, bool)

```

DEFINE
  canonicalizeSpecifier(Char)
    t(•, char)
DEFINE
  canonicalizeSpecifier(Signed Char)
    t(•, signed-char)
DEFINE
  canonicalizeSpecifier(Unsigned Char)
    t(•, unsigned-char)
RULE
  atomic(—)
    •
    [anywhere]
RULE
  alignas(—)
    •
    [anywhere]
DEFINE
  canonicalizeSpecifier(Double)
    t(•, double)
DEFINE
  canonicalizeSpecifier(Float)
    t(•, float)
DEFINE
  canonicalizeSpecifier(Long Double)
    t(•, long-double)
DEFINE
  canonicalizeSpecifier(Complex Double)
    t(Complex, double)
DEFINE
  canonicalizeSpecifier(Complex Float)
    t(Complex, float)

```

```

DEFINE
  canonicalizeSpecifier(Complex Long Double)
    t(Complex, long-double)
DEFINE
  canonicalizeSpecifier(Imaginary Double)
    t(Imaginary, double)
DEFINE
  canonicalizeSpecifier(Imaginary Float)
    t(Imaginary, float)
DEFINE
  canonicalizeSpecifier(Imaginary Long Double)
    t(Imaginary, long-double)
DEFINE
  canonicalizeSpecifier(B)
    t(•, short-int)
    when  $\left( \left( \begin{array}{l} (B ==_{Bag} \text{Short}) \\ \vee_{Bool} (B ==_{Bag} \text{Signed Short}) \\ \vee_{Bool} (B ==_{Bag} \text{Short Int}) \\ \vee_{Bool} (B ==_{Bag} \text{Signed Short Int}) \end{array} \right) \right)$ 
DEFINE
  canonicalizeSpecifier(B)
    t(•, unsigned-short-int)
    when  $\vee_{Bool} \left( \begin{array}{l} (B ==_{Bag} \text{Unsigned Short}) \\ (B ==_{Bag} \text{Unsigned Short Int}) \end{array} \right)$ 
DEFINE
  canonicalizeSpecifier(B)
    t(•, int)
    when  $\left( \begin{array}{l} (B ==_{Bag} \text{Int}) \\ \vee_{Bool} (B ==_{Bag} \text{Signed}) \\ \vee_{Bool} (B ==_{Bag} \text{Signed Int}) \end{array} \right)$ 
DEFINE
  canonicalizeSpecifier(B)
    t(•, unsigned-int)
    when  $\vee_{Bool} \left( \begin{array}{l} (B ==_{Bag} \text{Unsigned}) \\ (B ==_{Bag} \text{Unsigned Int}) \end{array} \right)$ 

```

```

DEFINE
  canonicalizeSpecifier( $B$ )
  -----
  t( $\bullet$ , long-int)
  when  $\left( \left( \begin{array}{l} (B ==_{Bag} \text{ Long}) \\ \vee_{Bool} (B ==_{Bag} \text{ Signed Long}) \\ \vee_{Bool} (B ==_{Bag} \text{ Long Int}) \\ \vee_{Bool} (B ==_{Bag} \text{ Signed Long Int}) \end{array} \right) \right)$ 
DEFINE
  canonicalizeSpecifier( $B$ )
  -----
  t( $\bullet$ , unsigned-long-int)
  when  $\vee_{Bool} (B ==_{Bag} \text{ Unsigned Long Int})$ 
DEFINE
  canonicalizeSpecifier( $B$ )
  -----
  t( $\bullet$ , long-long-int)
  when  $\left( \left( \begin{array}{l} (B ==_{Bag} \text{ Long Long}) \\ \vee_{Bool} (B ==_{Bag} \text{ Signed Long Long}) \\ \vee_{Bool} (B ==_{Bag} \text{ Long Long Int}) \\ \vee_{Bool} (B ==_{Bag} \text{ Signed Long Long Int}) \end{array} \right) \right)$ 
DEFINE
  canonicalizeSpecifier( $B$ )
  -----
  t( $\bullet$ , unsigned-long-long-int)
  when  $\vee_{Bool} (B ==_{Bag} \text{ Unsigned Long Long Int})$ 
END MODULE

```

MODULE COMMON-SEMANTICS-TYPE-INTERPRETATION

IMPORTS COMMON-SEMANTICS-TYPE-INCLUDE

SYNTAX $K ::= \text{BitFieldType}(K, K)$ [strict]

RULE

$$\frac{\text{Specifier}(\text{List}(L))}{\text{canonicalizeType}(\text{Bag } L)}$$

RULE

$$\frac{\text{BitFieldType}(T, N : -)}{t(\bullet, \text{bitfieldType}(T, N))}$$

[anywhere]

SYNTAX $KLabel ::= \text{makeArrayType}(\text{Nat})$ [function]
 | $\text{makeFunctionType}(\text{List}\{KResult\})$ [function]

SYNTAX $Type ::= \text{pushTypeDown}(Type, KLabel)$ [function]

SYNTAX $KLabel ::= \text{makePointerType}$ [function]
 | $\text{makeIncompleteArrayType}$ [function]

RULE

$$\frac{\text{ArrayType}(T, N : -, -)}{\text{pushTypeDown}(T, \text{makeArrayType}(N))}$$

when $N >_{Int} 0$
 [anywhere]

RULE

$$\frac{\text{ArrayType}(T, \text{emptyValue}, -)}{\text{pushTypeDown}(T, \text{makeIncompleteArrayType})}$$

[anywhere]

RULE

$$\frac{\text{PointerType}(T)}{\text{pushTypeDown}(T, \text{makePointerType})}$$

[anywhere]

RULE

$$\frac{\text{FunctionType}(T)}{T}$$

[anywhere]

RULE

$$\frac{\text{Prototype}(T, \text{List}(L), \text{false})}{\text{pushTypeDown}(T, \text{makeFunctionType}(L))}$$

[anywhere]

RULE

$$\frac{\text{Prototype}(T, \text{List}(L), \text{true})}{\text{pushTypeDown}(T, \text{makeFunctionType}(L, t(\bullet, \text{variadic})))}$$

[anywhere]

DEFINE **PUSHDOWN-ARRAY**

$$\frac{\text{pushTypeDown}(t(S, \text{arrayType}(T, N)), K)}{t(S, \text{arrayType}(\text{pushTypeDown}(T, K), N))}$$

DEFINE **PUSHDOWN-INCOMPLETE**

$$\frac{\text{pushTypeDown}(t(S, \text{incompleteArrayType}(T)), K)}{t(S, \text{incompleteArrayType}(\text{pushTypeDown}(T, K)))}$$

DEFINE **PUSHDOWN-POINTER**

$$\frac{\text{pushTypeDown}(t(S, \text{pointerType}(T)), K)}{t(S, \text{pointerType}(\text{pushTypeDown}(T, K)))}$$

DEFINE **PUSHDOWN-QUALIFIED**

$$\frac{\text{pushTypeDown}(t(S, \text{qualifiedType}(T, K)), K)}{t(S, \text{qualifiedType}(\text{pushTypeDown}(T, K), K))}$$

DEFINE **PUSHDOWN-FUNCTION**

$$\frac{\text{pushTypeDown}(t(S, \text{functionType}(T, L)), K)}{t(S, \text{functionType}(\text{pushTypeDown}(T, K), L))}$$

DEFINE **PUSHDOWN-STRUCT**

$$\frac{\text{pushTypeDown}(t(S, \text{structType}(X)), K)}{K(t(S, \text{structType}(X)))}$$

DEFINE **PUSHDOWN-UNION**

$$\frac{\text{pushTypeDown}(t(S, \text{unionType}(X)), K)}{K(t(S, \text{unionType}(X)))}$$

DEFINE **PUSHDOWN-ENUM**

$$\frac{\text{pushTypeDown}(t(S, \text{enumType}(X)), K)}{K(t(S, \text{enumType}(X)))}$$

DEFINE **PUSHDOWN-TYPEDEF**

$$\frac{\text{pushTypeDown}(t(S, \text{typedefType}(X, t(S', T))), K)}{K(t(S, S', T))}$$

```

DEFINE PUSHDOWN-BASIC
  
$$\frac{\text{pushTypeDown}(T, K)}{K(T)}$$

  when isBasicType( $T$ )

DEFINE
  
$$\frac{\text{makeArrayType}(N)(T)}{t(\bullet, \text{arrayType}(T, N))}$$

DEFINE
  
$$\frac{\text{makeFunctionType}(L)(T)}{t(\bullet, \text{functionType}(T, \text{giveNamesToArgs}(L)))}$$

DEFINE
  
$$\frac{\text{makePointerType}(T)}{t(\bullet, \text{pointerType}(T))}$$

DEFINE
  
$$\frac{\text{makeIncompleteArrayType}(T)}{t(\bullet, \text{incompleteArrayType}(T))}$$

SYNTAX  $List\{KResult\} ::= \text{giveNamesToArgs}(List\{KResult\})$  [function]
      |  $\text{giveNamesToArgs-aux}(Nat, List\{KResult\})$  [function]

DEFINE
  
$$\frac{\text{giveNamesToArgs}(L)}{\text{giveNamesToArgs-aux}(0, L)}$$

DEFINE
  
$$\frac{\text{giveNamesToArgs-aux}(N, \text{typedDecl}(T, X) \text{ , } L)}{\text{typedDecl}(T, X) \text{ , } \text{giveNamesToArgs-aux}(N, L)}$$

  when  $X \neq_K \#NoName$ 

DEFINE
  
$$\frac{\text{giveNamesToArgs-aux}(N, \text{typedDecl}(T, X) \text{ , } L)}{\text{typedDecl}(T, \#NoName(N)) \text{ , } \text{giveNamesToArgs-aux}(N +_{Int} 1, L)}$$

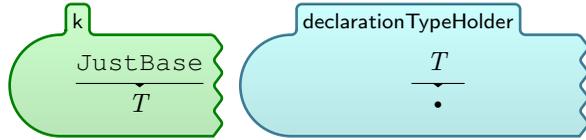
  when  $X ==_K \#NoName$ 

DEFINE
  
$$\frac{\text{giveNamesToArgs-aux}(\text{—}, t(\bullet, \text{variadic}))}{t(\bullet, \text{variadic})}$$


```

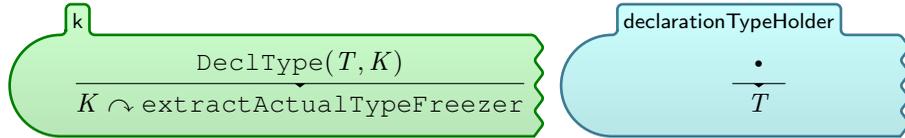
DEFINE
 $\underline{\text{giveNamesToArgs-aux}(-, \bullet)}$

RULE

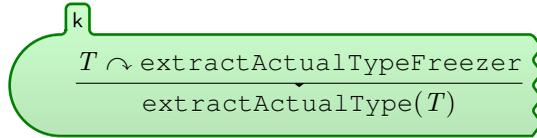


SYNTAX $K ::= \text{extractActualTypeFreezer}$

RULE



RULE



SYNTAX $List\{KResult\} ::= \text{fillUnionBitHoles}(List\{KResult\})$ [function]
 $| \text{fillUnionBitHoles-aux}(Nat, List\{KResult\})$ [function]

DEFINE

$\underline{\text{fillUnionBitHoles}(\bullet)}$

DEFINE

$\underline{\text{fillUnionBitHoles}(K, L)}$
 $\text{fillUnionBitHoles-aux}(0, K, L)$

DEFINE

$\underline{\text{fillUnionBitHoles-aux}(N, \text{typedDecl}(T, X), L)}$
 $\text{typedDecl}(T, X), \text{fillUnionBitHoles-aux}(N, L)$
 when $\neg_{Bool} \text{isBitFieldType}(T)$

DEFINE

$$\frac{\text{fillUnionBitHoles-aux}(N, \text{typedDecl}(t(S, \text{bitfieldType}(T, N')), X) \text{ , } L)}{\text{typedDecl}(t(S, \text{bitfieldType}(T, N')), X) \text{ , } \text{fillUnionBitHoles-aux}(\max \text{Int}(N, N'), L)}$$

DEFINE

$$\frac{\text{fillUnionBitHoles-aux}(N, \bullet)}{\text{typedDecl}(t(\bullet, \text{bitfieldType}(t(\bullet, \text{unsigned-int}), N +_{\text{Int}} ((\text{absInt}(\text{numBitsPerByte} -_{\text{Int}} (N \%_{\text{Int}} \text{numBitsPerByte}))) \%_{\text{Int}} \text{numBitsPerByte})), \# \text{NoName})}$$

SYNTAX $List\{KResult\} ::= \text{fillBitHoles}(List\{KResult\})$ [function]
| $\text{fillBitHoles-aux}(Nat, List\{KResult\})$ [function]

DEFINE **FILLBITHOLES-NONE**

$$\frac{\text{fillBitHoles}(\bullet)}{\bullet}$$

DEFINE **FILLBITHOLES-NOT-BITFIELD**

$$\frac{\text{fillBitHoles}(\text{typedDecl}(T, X) \text{ , } L)}{\text{typedDecl}(T, X) \text{ , } \text{fillBitHoles}(L)} \\ \text{when } \neg_{\text{Bool}} \text{isBitfieldType}(T)$$

DEFINE **FILLBITHOLES-BITFIELD**

$$\frac{\text{fillBitHoles}(\text{typedDecl}(T, X) \text{ , } L)}{\text{fillBitHoles-aux}(0, \text{typedDecl}(T, X) \text{ , } L)} \\ \text{when } \text{isBitfieldType}(T)$$

DEFINE **FILLBITHOLES-AUX-NOT-BITFIELD**

$$\frac{\text{fillBitHoles-aux}(N, \text{typedDecl}(T, X) \text{ , } L)}{\text{typedDecl}(t(\bullet, \text{bitfieldType}(t(\bullet, \text{unsigned-int}), (\text{absInt}(\text{numBitsPerByte} -_{\text{Int}} (N \%_{\text{Int}} \text{numBitsPerByte}))) \%_{\text{Int}} \text{numBitsPerByte})), \# \text{NoName}) \text{ , } \text{fillBitHoles}(\text{typedDecl}(T, X) \text{ , } L)} \\ \text{when } \neg_{\text{Bool}} \text{isBitfieldType}(T)$$

DEFINE **FILLBITHOLES-AUX-BITFIELD-NORMAL**

$$\frac{\text{fillBitHoles-aux}(N, \text{typedDecl}(t(S, \text{bitfieldType}(T, N')), X) \text{ , } L)}{\text{typedDecl}(t(S, \text{bitfieldType}(T, N')), X) \text{ , } \text{fillBitHoles-aux}(N +_{\text{Int}} N', L)} \\ \text{when } N' \neq_{\text{Int}} 0$$

DEFINE **FILLBITHOLES-BITFIELD-ZERO**

$$\frac{\text{fillBitHoles-aux}(N, \text{typedDecl}(t(\text{—}, \text{bitfieldType}(T, N')), \text{—}) \text{ , } L)}{\text{typedDecl}(t(\bullet, \text{bitfieldType}(t(\bullet, \text{unsigned-int}), (\text{absInt}(\text{numBitsPerByte} -_{\text{Int}} (N \%_{\text{Int}} \text{numBitsPerByte}))) \%_{\text{Int}} \text{numBitsPerByte})), \# \text{NoName}) \text{ , } \text{fillBitHoles}(L)} \\ \text{when } N' ==_{\text{Int}} 0$$

DEFINE **FILLBITHOLES-DONE**

$$\frac{\text{fillBitHoles-aux}(N, \bullet)}{\text{typedDecl}(t(\bullet, \text{bitfieldType}(t(\bullet, \text{unsigned-int}), (\text{absInt}(\text{numBitsPerByte} -_{Int} (N \%_{Int} \text{numBitsPerByte}))) \%_{Int} \text{numBitsPerByte})), \#NoName)}$$

RULE

$$\frac{\text{typedDecl}(t(-, \text{bitfieldType}(-, N)), \#NoName), \text{typedDecl}(t(-, \text{bitfieldType}(-, N')), \#NoName)}{\text{typedDecl}(t(\bullet, \text{bitfieldType}(t(\bullet, \text{unsigned-int}), N +_{Int} N')), \#NoName)}$$

[anywhere]

SYNTAX $List\{KResult\} ::= \text{incompleteToFlexibleArrayMember}(List\{KResult\})$ [function]

DEFINE

$$\frac{\text{incompleteToFlexibleArrayMember}(\text{typedDecl}(T, X), L)}{\text{typedDecl}(T, X), \text{incompleteToFlexibleArrayMember}(L)}$$

when $\neg_{Bool} \text{isIncompleteType}(T)$

DEFINE

$$\frac{\text{incompleteToFlexibleArrayMember}(\text{typedDecl}(t(S, \text{incompleteArrayType}(T)), X))}{\text{typedDecl}(t(S, \text{flexibleArrayType}(T)), X)}$$

DEFINE

$$\frac{\text{incompleteToFlexibleArrayMember}(\bullet)}{\bullet}$$

RULE

k

$$\frac{\text{StructDef}(X, List(L))}{\text{addStruct}(X, \text{fillBitHoles}(\text{incompleteToFlexibleArrayMember}(L))) \curvearrow \text{giveType}(X, t(\bullet, \text{structType}(X)))}$$

RULE

k

$$\frac{\text{UnionDef}(X, List(L))}{\text{addUnion}(X, \text{fillUnionBitHoles}(L)) \curvearrow \text{giveType}(X, t(\bullet, \text{unionType}(X)))}$$

RULE

$$\frac{\text{OnlyTypedef}(K)}{K \rightsquigarrow \text{discard}}$$

RULE

$$\frac{\text{NameAndType}(X, T)}{\text{typedDecl}(T, X)} \\ \text{[anywhere]}$$

DEFINE EXTRACT-BASIC

$$\frac{\text{extractActualType}(T)}{T} \\ \text{when isBasicType}(T)$$

DEFINE EXTRACT-ENUM

$$\frac{\text{extractActualType}(t(S, \text{enumType}(X)))}{t(S, \text{enumType}(X))}$$

DEFINE EXTRACT-STRUCT

$$\frac{\text{extractActualType}(t(S, \text{structType}(X)))}{t(S, \text{structType}(X))}$$

DEFINE EXTRACT-UNION

$$\frac{\text{extractActualType}(t(S, \text{unionType}(X)))}{t(S, \text{unionType}(X))}$$

DEFINE EXTRACT-ARRAY

$$\frac{\text{extractActualType}(t(S, \text{arrayType}(T, N)))}{t(S, \text{arrayType}(\text{extractActualType}(T), N))}$$

DEFINE EXTRACT-INCOMPLETEARRAY

$$\frac{\text{extractActualType}(t(S, \text{incompleteArrayType}(T)))}{t(S, \text{incompleteArrayType}(\text{extractActualType}(T)))}$$

DEFINE EXTRACT-BITFIELD

$$\frac{\text{extractActualType}(t(S, \text{bitfieldType}(T, N)))}{t(S, \text{bitfieldType}(\text{extractActualType}(T), N))}$$

DEFINE EXTRACT-FUNCTION

$$\frac{\text{extractActualType}(t(S, \text{functionType}(T, \text{List})))}{t(S, \text{functionType}(\text{extractActualType}(T), \text{List}))}$$

DEFINE **EXTRACT-POINTER**

$$\frac{\text{extractActualType}(t(S, \text{pointerType}(T)))}{t(S, \text{pointerType}(\text{extractActualType}(T)))}$$
 DEFINE **EXTRACT-QUALIFIED**

$$\frac{\text{extractActualType}(t(S, \text{qualifiedType}(T, K)))}{t(S, \text{qualifiedType}(\text{extractActualType}(T), K))}$$
 DEFINE **EXTRACT-TYPEDEF**

$$\frac{\text{extractActualType}(t(S, \text{typedefType}(_, t(S', T))))}{\text{extractActualType}(t(S, S', T))}$$

SYNTAX $K ::= \text{NameAndType}(K, K)$ [**strict(2)**]

RULE

$$\frac{\text{SingleName}(T, \text{Name}(X, K))}{\text{NameAndType}(X, \text{DeclType}(T, K))}$$
 [**anywhere**]

RULE **SEPARATE-FIELDGROUPS**

$$\frac{\text{FieldGroup}(K, \text{List}(C, C', L))}{\text{FieldGroup}(K, \text{List}(C)) \text{ , } \text{FieldGroup}(K, \text{List}(C', L))}$$
 [**anywhere**]

RULE

$$\frac{\text{FieldGroup}(T, \text{List}(\text{Name}(X, K)))}{\text{NameAndType}(X, \text{DeclType}(T, K))}$$
 [**anywhere**]

RULE

$$\frac{\text{FieldGroup}(T, \text{List}(\text{BitFieldName}(\text{Name}(X, K), \text{Size})))}{\text{NameAndType}(X, \text{DeclType}(T, \text{BitFieldType}(K, \text{Size})))}$$
 [**anywhere**]

RULE

$$\frac{\text{FieldName}(K)}{K}$$
 [**anywhere**]

END MODULE

MODULE COMMON-SEMANTICS-TYPE-MISC

IMPORTS COMMON-SEMANTICS-TYPE-INCLUDE

(n1570) §6.2.7 ¶1 Two types have compatible type if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators. Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are completed anywhere within their respective translation units, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types; if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier; and if one member of the pair is declared with a name, the other is declared with the same name. For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values.

(n1570) §6.7.3 ¶10 For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

DEFINE TYPECOMPATIBLE-IDENTICAL

$\frac{\text{isTypeCompatible}(T, T)}{\text{true}}$

DEFINE TYPECOMPATIBLE-TWO-INTS

$\frac{\text{isTypeCompatible}(T, T')}{\text{true}}$

when hasIntegerType(T) \wedge_{Bool} hasIntegerType(T')

DEFINE TYPECOMPATIBLE-TWO-PTR

$\frac{\text{isTypeCompatible}(t(-, \text{pointerType}(-)), t(-, \text{pointerType}(-)))}{\text{true}}$

DEFINE TYPECOMPATIBLE-PTR-INT

$\frac{\text{isTypeCompatible}(t(-, \text{pointerType}(-)), T)}{\text{true}}$

when hasIntegerType(T)

```

DEFINE TYPECOMPATIBLE-INT-PTR
  isTypeCompatible( $T, t(-, \text{pointerType}(-))$ )
  
    true
  
  when hasIntegerType( $T$ )

DEFINE TYPECOMPATIBLE-DECLARATIONS
  isTypeCompatible( $\text{typedDecl}(T, -), \text{typedDecl}(T', -)$ )
  
    isTypeCompatible( $T, T'$ )
  

DEFINE TYPECOMPATIBLE-PROTOTYPES
  isTypeCompatible( $t(-, \text{prototype}(T)), t(-, \text{prototype}(T'))$ )
  
    isTypeCompatible( $T, T'$ )
  

DEFINE TYPECOMPATIBLE-ARRAY-RIGHT
  isTypeCompatible( $T, t(S, \text{arrayType}(T', -))$ )
  
    isTypeCompatible( $T, t(S, \text{pointerType}(T'))$ )
  

DEFINE TYPECOMPATIBLE-ARRAY-LEFT
  isTypeCompatible( $t(S, \text{arrayType}(T, -)), T'$ )
  
    isTypeCompatible( $t(S, \text{pointerType}(T)), T'$ )
  

DEFINE TYPECOMPATIBLE-INCOMPLETEARRAY-RIGHT
  isTypeCompatible( $T, t(S, \text{incompleteArrayType}(T'))$ )
  
    isTypeCompatible( $T, t(S, \text{pointerType}(T'))$ )
  

DEFINE TYPECOMPATIBLE-INCOMPLETEARRAY-LEFT
  isTypeCompatible( $t(S, \text{incompleteArrayType}(T)), T'$ )
  
    isTypeCompatible( $t(S, \text{pointerType}(T)), T'$ )
  

DEFINE TYPECOMPATIBLE-FUNCTION-VOID-LEFT
  isTypeCompatible( $t(-, \text{functionType}(T_1, \text{typedDecl}(t(-, \text{void}), -))), t(-, \text{functionType}(T_2, \bullet))$ )
  
    isTypeCompatible( $T_1, T_2$ )
  

DEFINE TYPECOMPATIBLE-FUNCTION-VOID-RIGHT
  isTypeCompatible( $t(-, \text{functionType}(T_1, \bullet)), t(-, \text{functionType}(T_2, \text{typedDecl}(t(-, \text{void}), -)))$ )
  
    isTypeCompatible( $T_1, T_2$ )
  

DEFINE TYPECOMPATIBLE-FUNCTION
  isTypeCompatible( $t(S, \text{functionType}(T_1, T', L)), t(S', \text{functionType}(T_2, T'', L'))$ )
  
    isTypeCompatible( $t(S, \text{functionType}(T_1, L)), t(S', \text{functionType}(T_2, L')) \wedge_{Bool} \text{isTypeCompatible}(T', T'')$ )
  

```

DEFINE `TYPECOMPATIBLE-INCOMPLETEARRAY-NIL`

$$\frac{\text{isTypeCompatible}(t(-, \text{functionType}(T_1, \bullet)), t(-, \text{functionType}(T_2, \bullet)))}{\text{isTypeCompatible}(T_1, T_2)}$$

DEFINE

$$\frac{\text{true}}{\text{isTypeCompatible}(T, T')}$$

 when $\left(\frac{\text{hasIntegerType}(T)}{\vee_{Bool} \text{isFloatType}(T)} \right) \wedge_{Bool} \left(\frac{\text{hasIntegerType}(T')}{\vee_{Bool} \text{isFloatType}(T')} \right)$

SYNTAX $K ::= \text{addGlobalAggregate}(Id, K)$
 $\quad | \text{addLocalAggregate}(Id, K)$
 $\quad | \text{addStruct- aux}(Id, List\{KResult\}, K, Map, Map, List\{KResult\})$ [strict(3)]
 $\quad | \text{addUnion- aux}(Id, List\{KResult\}, Map, Map, List\{KResult\})$

RULE

k

$$\frac{\text{addStruct}(S, L)}{\text{addStruct- aux}(S, L, 0 : \text{cfg:largestUnsigned}, \bullet, \bullet, L)}$$

 when $L \neq_{List\{K\}} \bullet$

RULE

k

$$\frac{\text{addStruct- aux}(S, \text{typedDecl}(T, Field), L, V, Types, Offsets, L')}{\text{addStruct- aux}(S, L, V + \text{bitSizeofType}(T), Types [T / Field], Offsets [\text{value}(V) / Field], L')}$$

RULE

k

$$\frac{\text{addStruct- aux}(S, \bullet, -, Types, Offsets, L)}{\text{if } F ==_K \text{ File-Scope}$$

 then $\text{addGlobalAggregate}(S, \text{aggregateInfo}(L, Types, Offsets))$
 else
 $\text{addLocalAggregate}(S, \text{aggregateInfo}(L, Types, Offsets))$
 fi

currentFunction
 F

RULE

$$\frac{\text{addUnion}(S, L)}{\text{addUnion-aux}(S, L, \bullet, \bullet, L)}$$

when $L \neq_{List} \{K\} \bullet$

RULE

$$\frac{\text{addUnion-aux}(S, \text{typedDecl}(T, Field), L, Types, Offsets, L')}{\text{addUnion-aux}(S, L, Types [T / Field], Offsets [0 / Field], L')}$$

RULE

$$\frac{\text{addUnion-aux}(S, \bullet, Types, Offsets, L)}{\begin{array}{l} \text{if } F ==_K \text{ File-Scope} \\ \text{then } \text{addGlobalAggregate}(S, \text{aggregateInfo}(L, Types, Offsets)) \\ \text{else} \\ \quad \text{addLocalAggregate}(S, \text{aggregateInfo}(L, Types, Offsets)) \\ \text{fi} \end{array}}$$

currentFunction
 F

RULE

$$\frac{\text{addGlobalAggregate}(X, K)}{\bullet}$$

$$\frac{\text{structs } M'}{M' [K / X]}$$

$$\frac{\text{gstructs } M}{M [K / X]}$$

RULE

$$\frac{\text{addLocalAggregate}(X, K)}{\bullet}$$

$$\frac{\text{structs } M}{M [K / X]}$$

DEFINE

$$\frac{\text{isTypeResult}(t(-, T))}{\text{true}}$$

when setOfTypes contains l(getKLabel(T))

```

DEFINE
  isTypeResult(T)
  true
  when isBasicType(T)
DEFINE
  isTypeResult(K)
  false
  when (getKLabel(K) !=KLabel t
DEFINE
  isTypeResult(t(S, T))
  false
  when (¬Bool (setOfTypes contains l(getKLabel(T)))) ∧Bool (¬Bool isBasicType(t(S, T)))
DEFINE
  isFloatType(t(−, float))
  true
DEFINE
  isFloatType(t(−, double))
  true
DEFINE
  isFloatType(t(−, long-double))
  true
DEFINE
  isFloatType(t(−, T))
  false
  when (((T !=K float) ∧Bool (T !=K double)) ∧Bool (T !=K long-double)) ∧Bool ((getKLabel(T)) !=KLabel qualifiedType)
DEFINE ISCHARTYPE-CHAR
  isCharType(t(−, char))
  true
DEFINE ISCHARTYPE-QUALIFIED
  isCharType(t(−, qualifiedType(T, −)))
  isCharType(T)

```

DEFINE ISCHARTYPE-UNSIGNED-CHAR
isCharType(t(—, unsigned-char))

true

DEFINE ISCHARTYPE-SIGNED-CHAR
isCharType(t(—, signed-char))

true

DEFINE ISCHARTYPE-OTHER
isCharType(t(—, T))

false

when (((T \neq_K char) \wedge_{Bool} (T \neq_K unsigned-char)) \wedge_{Bool} (T \neq_K signed-char)) \wedge_{Bool} ((getKLabel(T)) \neq_{KLabel} qualifiedType)

DEFINE ISWCHARTYPE-WCHAR
isWCharType(t(—, T))

true

when T \equiv_K simpleType(cfg:wcharut)

DEFINE ISWCHARTYPE-OTHER
isWCharType(t(—, T))

false

when (T \neq_K simpleType(cfg:wcharut)) \wedge_{Bool} ((getKLabel(T)) \neq_{KLabel} qualifiedType)

DEFINE ISWCHARTYPE-QUALIFIED
isWCharType(t(—, qualifiedType(T, —)))

isWCharType(T)

DEFINE ISPOINTERTYPE-POINTER
isPointerType(t(—, pointerType(—)))

true

DEFINE ISPOINTERTYPE-QUALIFIED
isPointerType(t(—, qualifiedType(T, —)))

isPointerType(T)

DEFINE ISPOINTERTYPE-OTHER
isPointerType(t(—, T))

false

when ((getKLabel(T)) \neq_{KLabel} pointerType) \wedge_{Bool} ((getKLabel(T)) \neq_{KLabel} qualifiedType)

DEFINE ISBOOLTYPE-BOOL

$$\frac{\text{isBoolType}(t(-, \text{bool}))}{\text{true}}$$

DEFINE ISBOOLTYPE-QUALIFIEDTYPE

$$\frac{\text{isBoolType}(t(-, \text{qualifiedType}(T, -)))}{\text{isBoolType}(T)}$$

DEFINE ISBOOLTYPE-OTHER

$$\frac{\text{isBoolType}(t(-, T))}{\text{false}}$$
 when $(T \neq_K \text{bool}) \wedge_{Bool} ((\text{getKLabel}(T)) \neq_{KLabel} \text{qualifiedType})$

DEFINE ISARRAYTYPE-ARRAY

$$\frac{\text{isArrayType}(t(-, \text{arrayType}(-, -)))}{\text{true}}$$

DEFINE ISARRAYTYPE-INCOMPLETEARRAY

$$\frac{\text{isArrayType}(t(-, \text{incompleteArrayType}(-)))}{\text{true}}$$

DEFINE ISARRAYTYPE-FLEXIBLEARRAY

$$\frac{\text{isArrayType}(t(-, \text{flexibleArrayType}(-)))}{\text{true}}$$

DEFINE ISARRAYTYPE-QUALIFIED

$$\frac{\text{isArrayType}(t(-, \text{qualifiedType}(T, -)))}{\text{isArrayType}(T)}$$

DEFINE ISARRAYTYPE-OTHER

$$\frac{\text{isArrayType}(t(-, T))}{\text{false}}$$
 when $((((\text{getKLabel}(T)) \neq_{KLabel} \text{arrayType}) \wedge_{Bool} ((\text{getKLabel}(T)) \neq_{KLabel} \text{incompleteArrayType})) \wedge_{Bool} ((\text{getKLabel}(T)) \neq_{KLabel} \text{flexibleArrayType})) \wedge_{Bool} ((\text{getKLabel}(T)) \neq_{KLabel} \text{qualifiedType}))$

DEFINE

$$\frac{\text{isAggregateType}(T)}{\text{isArrayType}(T) \vee_{Bool} \text{isStructType}(T)}$$

\wedge_{Bool}

```

DEFINE ISSTRUCTTYPE-STRUCT
  isStructType(t(—, structType(—)))
  true
DEFINE ISSTRUCTTYPE-QUALIFIED
  isStructType(t(—, qualifiedType(T, —)))
  isStructType(T)
DEFINE ISSTRUCTTYPE-OTHER
  isStructType(t(—, T))
  false
  when ((getKLabel(T)) /=KLabel structType) ∧Bool ((getKLabel(T)) /=KLabel qualifiedType)
DEFINE ISUNIONTYPE-UNION
  isUnionType(t(—, unionType(—)))
  true
DEFINE ISUNIONTYPE-QUALIFIED
  isUnionType(t(—, qualifiedType(T, —)))
  isUnionType(T)
DEFINE ISUNIONTYPE-OTHER
  isUnionType(t(—, T))
  false
  when ((getKLabel(T)) /=KLabel unionType) ∧Bool ((getKLabel(T)) /=KLabel qualifiedType)
DEFINE ISINCOMPLETETYPE-TRUE
  isIncompleteType(t(—, incompleteArrayType(—)))
  true
DEFINE ISINCOMPLETETYPE-QUALIFIED
  isIncompleteType(t(—, qualifiedType(T, —)))
  isIncompleteType(T)
DEFINE ISINCOMPLETETYPE-FALSE
  isIncompleteType(t(—, T))
  false
  when ((getKLabel(T)) /=KLabel incompleteArrayType) ∧Bool ((getKLabel(T)) /=KLabel qualifiedType)

```

```

DEFINE ISEXTERNTYPE-QUALIFIED
  
$$\frac{\text{isExternType}(t(-, \text{qualifiedType}(T, K)))}{\text{if } K ==_K \text{ Extern} \\ \text{then true} \\ \text{else} \\ \text{isExternType}(T) \\ \text{fi}}$$

DEFINE ISEXTERNTYPE-FALSE
  
$$\frac{\text{isExternType}(t(-, T))}{\text{false}}$$

  when (getKLabel( $T$ ))  $\neq_{KLabel}$  qualifiedType
DEFINE ISSTATICTYPE-QUALIFIED
  
$$\frac{\text{isStaticType}(t(-, \text{qualifiedType}(T, K)))}{\text{if } K ==_K \text{ Static} \\ \text{then true} \\ \text{else} \\ \text{isStaticType}(T) \\ \text{fi}}$$

DEFINE ISSTATICTYPE-FALSE
  
$$\frac{\text{isStaticType}(t(-, T))}{\text{false}}$$

  when (getKLabel( $T$ ))  $\neq_{KLabel}$  qualifiedType
DEFINE ISCONSTTYPE-QUALIFIED
  
$$\frac{\text{isConstType}(t(-, \text{qualifiedType}(T, K)))}{\text{isConstType}(T)}$$

DEFINE ISCONSTTYPE-FALSE
  
$$\frac{\text{isConstType}(t(S, T))}{\text{false}}$$

  when ((getKLabel( $T$ ))  $\neq_{KLabel}$  qualifiedType)  $\wedge_{Bool}$  ( $\neg_{Bool}$  (Const in  $S$ ))
DEFINE ISCONSTTYPE-TRUE
  
$$\frac{\text{isConstType}(t(\text{Const } -, T))}{\text{true}}$$


```

```

DEFINE ISBITFIELDTYPE-TRUE
  isBitFieldType(t(—,bitFieldType(—,—))
    true
DEFINE ISBITFIELDTYPE-FALSE
  isBitFieldType(t(—, T))
    false
  when (getKLabel(T))  $\neq_{KLabel}$  bitFieldType
DEFINE ISFUNCTIONTYPE-TRUE
  isFunctionType(t(—,functionType(—,—))
    true
DEFINE ISFUNCTIONTYPE-PROTOTYPE
  isFunctionType(t(—,prototype(T))
    isFunctionType(T)
DEFINE ISFUNCTIONTYPE-QUALIFIED
  isFunctionType(t(—,qualifiedType(T,—))
    isFunctionType(T)
DEFINE ISFUNCTIONTYPE-FALSE
  isFunctionType(t(—, T))
    false
  when (((getKLabel(T))  $\neq_{KLabel}$  functionType)  $\wedge_{Bool}$  ((getKLabel(T))  $\neq_{KLabel}$  qualifiedType))  $\wedge_{Bool}$  ((getKLabel(T))  $\neq_{KLabel}$  prototype)
DEFINE ISFUNCTIONPOINTERTYPE-FP
  isFunctionPointerType(t(—,pointerType(t(—,functionType(—,—))))
    true
DEFINE ISFUNCTIONPOINTERTYPE-QUALIFIED
  isFunctionPointerType(t(—,qualifiedType(T,—))
    isFunctionPointerType(T)
DEFINE ISFUNCTIONPOINTERTYPE-QUALIFIED-POINTER
  isFunctionPointerType(t(—,pointerType(t(—,qualifiedType(T,—))))
    isFunctionPointerType(t(•,pointerType(T)))

```

```

DEFINE ISFUNCTIONPOINTERTYPE-NOTPOINTER
  isFunctionPointerType( $t(-, T)$ )
   $\downarrow$ 
  false
  when ((getKLabel( $T$ ))  $\neq_{KLabel}$  pointerType)  $\wedge_{Bool}$  ((getKLabel( $T$ ))  $\neq_{KLabel}$  qualifiedType)

```

```

DEFINE ISFUNCTIONPOINTERTYPE-NOTFUNCTION
  isFunctionPointerType( $t(-, \text{pointerType}(t(-, T)))$ )
   $\downarrow$ 
  false
  when ((getKLabel( $T$ ))  $\neq_{KLabel}$  functionType)  $\wedge_{Bool}$  ((getKLabel( $T$ ))  $\neq_{KLabel}$  qualifiedType)

```

```

DEFINE
  isArithmeticType( $T$ )
   $\downarrow$ 
  hasIntegerType( $T$ )
   $\vee_{Bool}$  isFloatType( $T$ )

```

```

DEFINE
  unqualifyType( $t(-, \text{qualifiedType}(T, -))$ )
   $\downarrow$ 
   $T$ 

```

```

DEFINE
  unqualifyType( $t(-, T)$ )
   $\downarrow$ 
   $t(\bullet, T)$ 
  when (getKLabel( $T$ ))  $\neq_{KLabel}$  qualifiedType

```

```

DEFINE
  removeStorageSpecifiers( $t(-, \text{qualifiedType}(T, -))$ )
   $\downarrow$ 
   $T$ 

```

```

DEFINE
  removeStorageSpecifiers( $t(S, T)$ )
   $\downarrow$ 
   $t(S, T)$ 
  when (getKLabel( $T$ ))  $\neq_{KLabel}$  qualifiedType

```

```

DEFINE
  getModifiers( $t(S, -)$ )
   $\downarrow$ 
   $S$ 

```

END MODULE

MODULE COMMON-C-TYPING

```

IMPORTS COMMON-SEMANTICS-TYPE-INCLUDE

IMPORTS COMMON-SEMANTICS-TYPE-DECLARATIONS

IMPORTS COMMON-SEMANTICS-TYPE-CANONICALIZATION

IMPORTS COMMON-SEMANTICS-TYPE-INTERPRETATION

IMPORTS COMMON-SEMANTICS-TYPE-MISC

```

```
END MODULE
```

```
MODULE DYNAMIC-SEMANTICS-TYPE-INCLUDE
```

```
  IMPORTS DYNAMIC-INCLUDE
```

```
  IMPORTS COMMON-SEMANTICS-TYPE-INCLUDE
```

```
END MODULE
```

```
MODULE DYNAMIC-SEMANTICS-TYPE-STRICTNESS
```

```
  IMPORTS DYNAMIC-SEMANTICS-TYPE-INCLUDE
```

```
  SYNTAX  $K ::= \text{waitingOnDeclType}$ 
```

```
  RULE TYPE-CAST-HEAT
```



```
  RULE TYPE-CAST-COOL
```

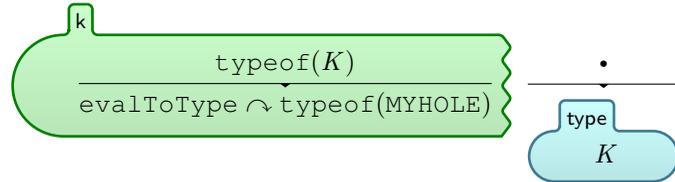


```
END MODULE
```

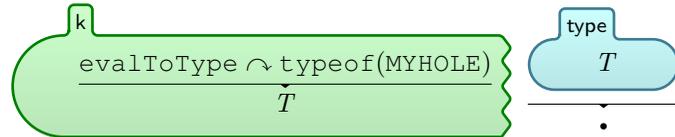
MODULE DYNAMIC-SEMANTICS-TYPE-EXPRESSIONS

IMPORTS DYNAMIC-SEMANTICS-TYPE-INCLUDE

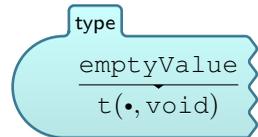
RULE



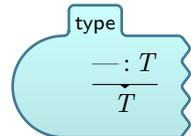
RULE



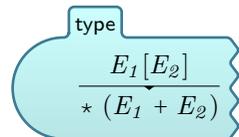
RULE



RULE



RULE



235

(n1570) §6.5.7 ¶13 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. . .

RULE

$$\frac{\text{type } T \ll -}{\text{promote}(T)}$$

RULE

$$\frac{\text{type } T \gg -}{\text{promote}(T)}$$

(n1570) §6.5.3.4 ¶5 The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is `size_t`, defined in `<stddef.h>` (and other headers).

RULE

$$\frac{\text{type } \text{SizeofExpression}(-)}{\text{cfg:sizeut}}$$

RULE

$$\frac{\text{type } t(S, \text{pointerType}(T)) + T'}{t(S, \text{pointerType}(T))}$$

when `hasIntegerType(T')`

RULE

$$\frac{\text{type } T' + t(S, \text{pointerType}(T))}{t(S, \text{pointerType}(T))}$$

when `hasIntegerType(T')`

RULE

$$\frac{\text{type} \quad t(S, \text{pointerType}(T)) - T'}{t(S, \text{pointerType}(T))}$$

when hasIntegerType(T')

RULE

$$\frac{\text{type} \quad t(-, \text{pointerType}(T)) - t(-, \text{pointerType}(T'))}{\text{cfg:ptrdiffut}}$$

RULE

$$\frac{\text{type} \quad t(S, \text{arrayType}(T, -)) + T'}{t(S, \text{pointerType}(T))}$$

when hasIntegerType(T')

RULE

$$\frac{\text{type} \quad T' + t(S, \text{arrayType}(T, -))}{t(S, \text{pointerType}(T))}$$

when hasIntegerType(T')

RULE

$$\frac{\text{type} \quad t(S, \text{arrayType}(T, -)) - T'}{t(S, \text{pointerType}(T))}$$

when hasIntegerType(T')

RULE

$$\frac{\text{type} \quad \text{Constant}(\text{StringLiteral}(S))}{t(\bullet, \text{arrayType}(t(\bullet, \text{char}), \text{lengthString}(S) +_{Int} 1))}$$

RULE

$$\frac{\text{type} \quad \text{Constant}(\text{WStringLiteral}(L))}{t(\bullet, \text{arrayType}(\text{cfg:wcharut}, (\text{length}_{\text{List}K}(L)) + \text{Int } 1))}$$

RULE

$$\frac{\text{type} \quad K}{T} \quad \text{types} \quad K \mapsto T$$

(n1570) §6.5.17 ¶2 The left operand of a comma operator is evaluated as a void expression; there is a sequence point between its evaluation and that of the right operand. Then the right operand is evaluated; the result has its type and value.

RULE

$$\frac{\text{type} \quad \text{Comma}(\text{List}(-, K))}{K}$$

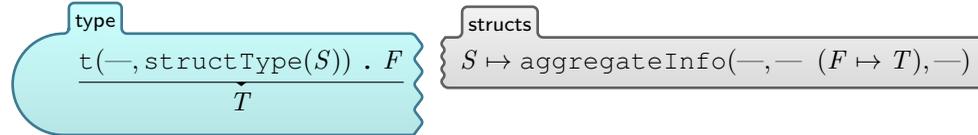
(n1570) §6.5.2.2 ¶5 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type **void**.

RULE TYPE-CALL-FUNC

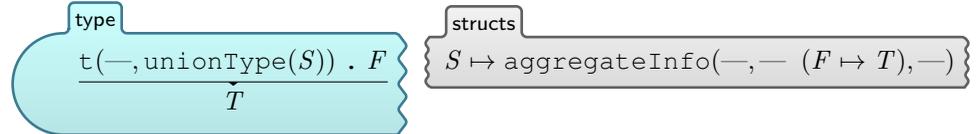
$$\frac{\text{type} \quad \text{Call}(T, -)}{\text{innerType}(T)}$$

(n1570) §6.5.2.3 ¶3 A postfix expression followed by the . operator and an identifier designates a member of a structure or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

RULE TYPE-STRUCT-DOT

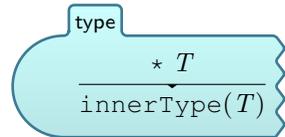


RULE TYPE-UNION-DOT

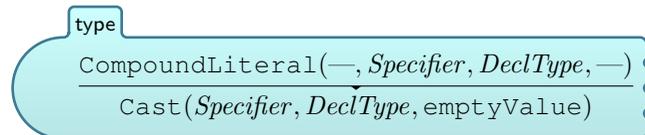


(n1570) §6.5.3.2 ¶4 The unary * operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type “pointer to type”, the result has type “type”. ...

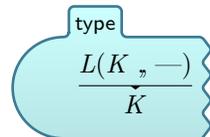
RULE TYPE-DEREF-TYPE



RULE TYPE-COMPOUND-LITERAL



RULE TYPE-ASSIGNMENT



when assignmentLabels contains l(L)

RULE

$$\frac{\text{type} \quad L(T, T')}{\text{usualArithmeticConversion}(T, T')}$$

when $\left(\text{isArithBinConversionOp}(L) \wedge_{Bool} \left(\text{hasIntegerType}(T) \right) \right) \wedge_{Bool} \left(\text{hasIntegerType}(T') \right)$
 $\left(\forall_{Bool} \text{isFloatType}(T) \right) \wedge_{Bool} \left(\forall_{Bool} \text{isFloatType}(T') \right)$

RULE TYPE-TERNARY-ARITHMETIC

$$\frac{\text{type} \quad - ? T : T'}{\text{usualArithmeticConversion}(T, T')}$$

when $\left(\text{hasIntegerType}(T) \right) \wedge_{Bool} \left(\text{hasIntegerType}(T') \right)$
 $\left(\forall_{Bool} \text{isFloatType}(T) \right) \wedge_{Bool} \left(\forall_{Bool} \text{isFloatType}(T') \right)$

RULE TYPE-TERNARY-IDENTICAL

$$\frac{\text{type} \quad - ? T : T}{T}$$

when $\neg_{Bool} \text{isArrayType}(T)$

RULE TYPE-TERNARY-ARRAY-LEFT

$$\frac{\text{type} \quad - ? t(S, \text{arrayType}(T, -)) : -}{t(S, \text{pointerType}(T))}$$

RULE TYPE-TERNARY-ARRAY-RIGHT

$$\frac{\text{type} \quad - ? - : t(S, \text{arrayType}(T, -))}{t(S, \text{pointerType}(T))}$$

RULE TYPE-TERNARY-POINTER

$$\frac{\text{type} \quad - ? t(S, \text{pointerType}(T)) : t(S', \text{pointerType}(T'))}{t(S, \text{pointerType}(T))}$$

RULE

$$\frac{\text{type} \quad L(T)}{T}$$

when $\text{isArithUnaryOp}(L) \wedge_{Bool} \text{isFloatType}(T)$

RULE

$$\frac{\text{type} \quad L(T)}{\text{promote}(T)}$$

when $\text{isArithUnaryOp}(L) \wedge_{Bool} \text{hasIntegerType}(T)$

RULE

$$\frac{\text{type} \quad L(T, _)}{T}$$

when $\text{isFloatType}(T) \wedge_{Bool} (\text{typeStrictLeftBinaryOperators} \text{ contains } l(L))$

RULE

$$\frac{\text{type} \quad L(T, _)}{\text{promote}(T)}$$

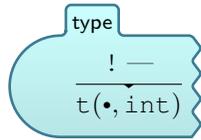
when $\text{hasIntegerType}(T) \wedge_{Bool} (\text{typeStrictLeftBinaryOperators} \text{ contains } l(L))$

RULE **TYPE-INC-DEC**

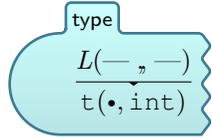
$$\frac{\text{type} \quad L(T)}{T}$$

when $\text{isPointerType}(T) \wedge_{Bool} \left(\left(\left(\begin{array}{l} (L == KLabel _++) \\ \vee_{Bool} (L == KLabel _--) \\ \vee_{Bool} (L == KLabel _--_) \\ \vee_{Bool} (L == KLabel _+_) \end{array} \right) \right) \right)$

RULE

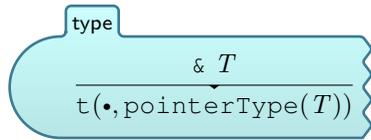


RULE



when $\left(\left(\begin{array}{l} (L == KLabel _ == _) \\ \vee_{Bool} (L == KLabel _ != _) \\ \vee_{Bool} (L == KLabel _ \& \& _) \\ \vee_{Bool} (L == KLabel _ || _) \end{array} \right) \right)$

RULE TYPE-ADDRESS

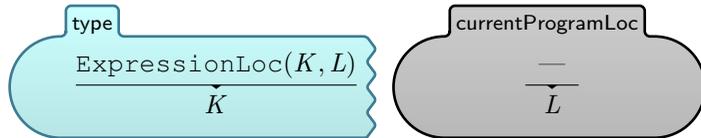


END MODULE

MODULE DYNAMIC-SEMANTICS-TYPE-MISC

IMPORTS DYNAMIC-SEMANTICS-TYPE-INCLUDE

RULE EXPRESSIONLOC-TYPE



END MODULE

MODULE DYNAMIC-C-TYPING

IMPORTS DYNAMIC-SEMANTICS-TYPE-INCLUDE

IMPORTS DYNAMIC-SEMANTICS-TYPE-STRICTNESS

```
IMPORTS DYNAMIC-SEMANTICS-TYPE-EXPRESSIONS
```

```
IMPORTS DYNAMIC-SEMANTICS-TYPE-MISC
```

```
END MODULE
```

A.6 Declarations

This section represents the static semantics of declarations. It handles both the processing of declarations, as well as the resolution or linking phase that occurs when combining multiple translation units (e.g., linking `.o` files).

MODULE COMMON-SEMANTICS-DECLARATIONS-INCLUDE

IMPORTS COMMON-INCLUDE

SYNTAX $K ::= \text{figureInit-aux}(Id, Type, K) \text{ [strict(3)]}$
| $\text{declObj}(Type, K, K)$
| external
| internal
| noLinkage

SYNTAX $KResult ::= \text{initializer}(K)$

SYNTAX $K ::= \text{startInit}(Type, Id, K)$
| $\text{doDeclare}(K, K) \text{ [strict(1)]}$
| $\text{processFunctionBody}(K)$

END MODULE

MODULE COMMON-SEMANTICS-DECLARATIONS-GENERAL

IMPORTS COMMON-SEMANTICS-DECLARATIONS-INCLUDE

SYNTAX $K ::= \text{defineType}(K) \text{ [strict]}$

CONTEXT: $\text{DeclarationDefinition}(\text{InitNameGroup}(\square, -))$

CONTEXT: $\text{Typedef}(\text{NameGroup}(\square, -))$

RULE

k
$$\frac{\text{figureInit}(X, T, \text{CodeLoc}(K, L))}{\text{CodeLoc}(\bullet, L) \rightsquigarrow \text{figureInit}(X, T, K)}$$

RULE

$$\frac{\text{figureInit}(X, T, \text{CompoundInit}(L))}{\text{giveType}(X, T) \rightsquigarrow \text{figureInit-aux}(X, T, \text{startInit}(T, X, \text{CompoundInit}(L)))}$$

when $\text{isAggregateType}(T)$
 $\vee_{Bool} \text{isUnionType}(T)$

RULE

$$\frac{\text{figureInit}(X, t(Se, \text{arrayType}(T, Len)), \text{SingleInit}(\text{Constant}(\text{StringLiteral}(S))))}{\text{CompoundInit}(\text{List}(\text{InitFragment}(\text{NextInit}, \text{SingleInit}(\text{Constant}(\text{StringLiteral}(S))))))}$$

when $\text{isCharType}(T) \wedge_{Bool} (\text{lengthString}(S) \leq_{Int} Len)$

RULE

$$\frac{\text{figureInit}(X, t(Se, \text{arrayType}(T, Len)), \text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S))))}{\text{CompoundInit}(\text{List}(\text{InitFragment}(\text{NextInit}, \text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S))))))}$$

when $\text{isWCharType}(T) \wedge_{Bool} ((\text{length}_{ListK}(S)) \leq_{Int} Len)$

RULE

$$\frac{\text{figureInit}(X, t(-, \frac{\text{incompleteArrayType}(T)}{\text{arrayType}(T, \text{lengthString}(S) +_{Int} 1)}, \text{SingleInit}(\text{Constant}(\text{StringLiteral}(S))))}{}$$

when $\text{isCharType}(T)$

RULE

$$\frac{\text{figureInit}(X, t(-, \frac{\text{incompleteArrayType}(T)}{\text{arrayType}(T, (\text{length}_{ListK}(S)) +_{Int} 1)}, \text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S))))}{}$$

when $\text{isWCharType}(T)$

RULE

$$\frac{\text{figureInit}(X, t(Se, incompleteArrayType(T)), CompoundInit(List(InitFragment(NextInit, SingleInit(Constant(StringLiteral(S)))))))}{\text{figureInit}(X, t(Se, incompleteArrayType(T)), SingleInit(Constant(StringLiteral(S))))}$$

RULE

$$\frac{\text{figureInit}(X, T, initializer(K))}{\text{figureInit-aux}(X, T, initializer(K))}$$

RULE

$$\frac{\text{figureInit-aux}(X, T, initializer(K))}{\text{initValue}(X, T, K)}$$

when $(\neg_{Bool} \text{isIncompleteType}(T)) \wedge_{Bool} (\neg_{Bool} \text{isConstType}(T))$

RULE

$$\frac{\text{figureInit-aux}(X, T, initializer(K))}{\text{initValue}(X, T, K \curvearrowright \text{makeUnwritableVar}(X))}$$

when $(\neg_{Bool} \text{isIncompleteType}(T)) \wedge_{Bool} \text{isConstType}(T)$

RULE

$$\frac{\text{figureInit}(X, T, SingleInit(K))}{\text{figureInit-aux}(X, T, initializer(\text{AllowWrite}(X) := K;))}$$

when $\neg_{Bool} \text{isArrayType}(T)$

RULE

$$\frac{\text{figureInit}(X, T, \text{CompoundInit}(\text{List}(\text{InitFragment}(\text{NextInit}, \text{SingleInit}(K))))))}{\text{figureInit-aux}(X, T, \text{initializer}(\text{AllowWrite}(X) := K;))}$$

when $\neg_{Bool} \left(\begin{array}{l} \text{isAggregateType}(T) \\ \vee_{Bool} \text{isUnionType}(T) \end{array} \right)$

RULE

$$\frac{\text{DeclarationDefinition}(\text{InitNameGroup}(T, \text{List}(K, K', L)))}{\text{DeclarationDefinition}(\text{InitNameGroup}(T, \text{List}(K))) \curvearrowright \text{DeclarationDefinition}(\text{InitNameGroup}(T, \text{List}(K', L)))}$$

RULE

$$\frac{\text{DeclarationDefinition}(\text{InitNameGroup}(T, \text{List}(\text{InitName}(Name, Exp))))}{\text{doDeclare}(\text{SingleName}(T, Name), Exp)}$$

RULE

$$\frac{\text{Typedef}(\text{NameGroup}(T, \text{List}(K, L)))}{\text{defineType}(\text{SingleName}(T, K)) \curvearrowright \text{Typedef}(\text{NameGroup}(T, \text{List}(L)))}$$

RULE

$$\frac{\text{Typedef}(\text{NameGroup}(T, \text{List}(\bullet)))}{\bullet}$$

RULE

$$\frac{\text{defineType}(\text{typedDecl}(T, X))}{\text{giveType}(\text{typedef}(X), T)}$$

SYNTAX

$K ::= \text{declareFunction}(Id, Type, K)$
 $\quad | \text{declareExternalVariable}(Id, Type, K)$
 $\quad | \text{declareInternalVariable}(Id, Type, K)$

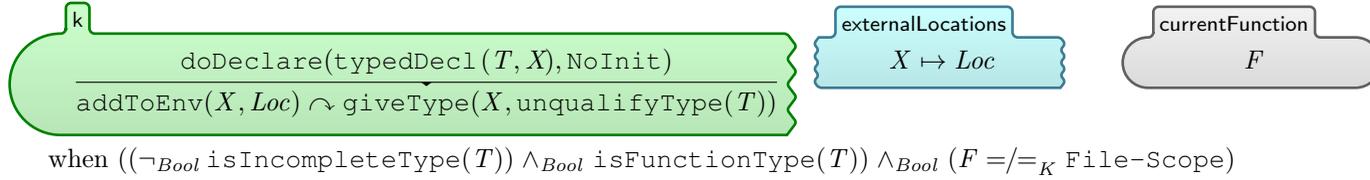
RULE

$$\frac{\text{doDeclare}(\text{typedDecl}(T, X), K)}{\text{declareFunction}(X, T, K)}$$

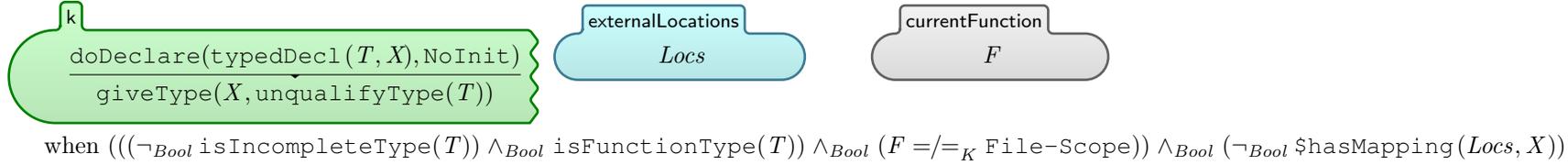
currentFunction
File-Scope

when $(\neg_{Bool} \text{isIncompleteType}(T)) \wedge_{Bool} \text{isFunctionType}(T)$

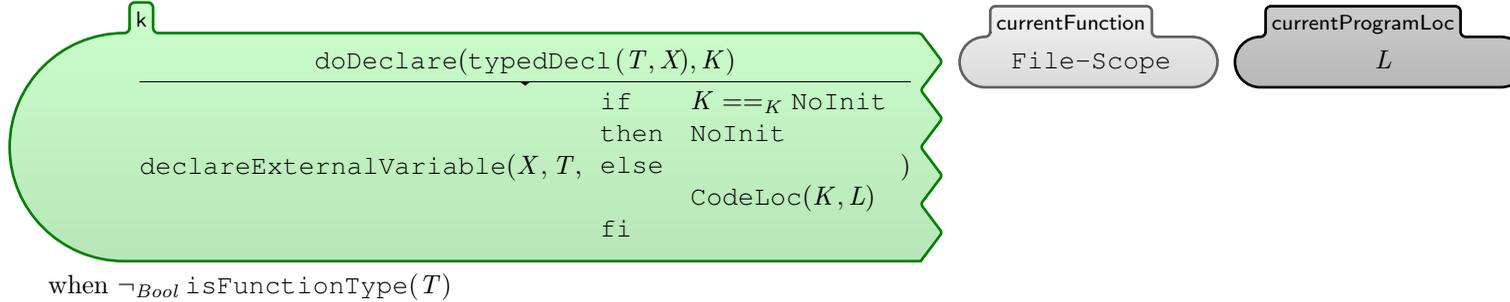
RULE



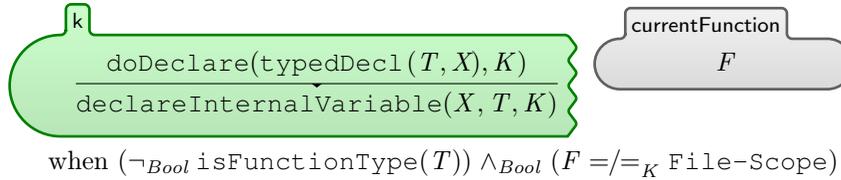
RULE



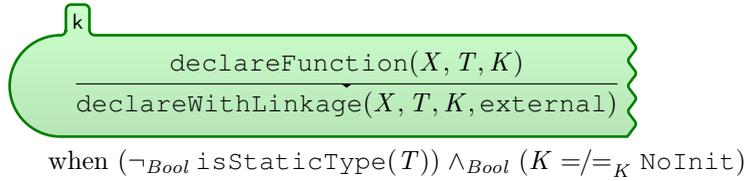
RULE



RULE



RULE



RULE

$$\frac{\text{declareFunction}(X, T, \text{NoInit})}{\text{declareWithLinkage}(X, t(\cdot, \text{prototype}(T)), \text{NoInit}, \text{external})}$$

when $\neg_{Bool} \text{isStaticType}(T)$

RULE

$$\frac{\text{declareFunction}(X, T, K)}{\text{declareWithLinkage}(X, T, K, \text{internal})}$$

when $\text{isStaticType}(T)$

currentFunction
File-Scope

SYNTAX $K ::= \text{declareWithLinkage}(Id, Type, K, K)$

RULE

$$\frac{\text{declareInternalVariable}(X, T, K)}{\text{declareWithLinkage}(X, T, K, \text{noLinkage})}$$

when $(\neg_{Bool} \text{isStaticType}(T)) \wedge_{Bool} (\neg_{Bool} \text{isExternType}(T))$

RULE

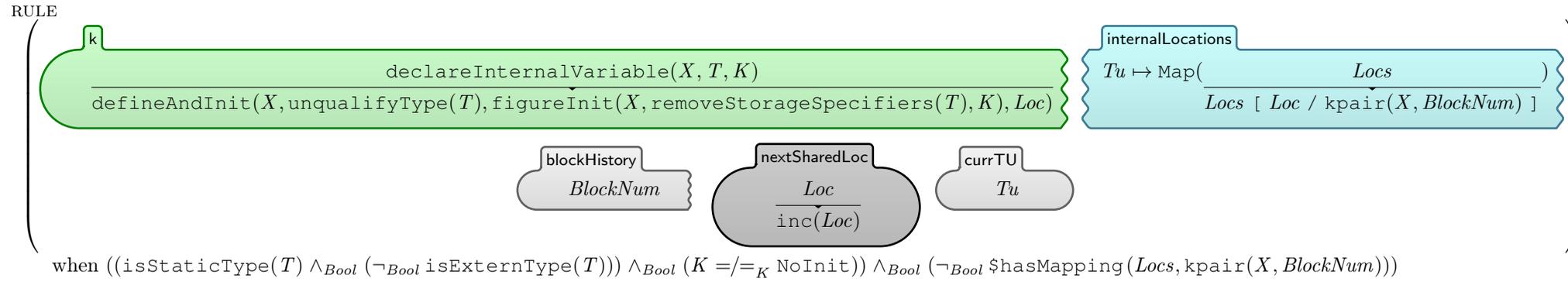
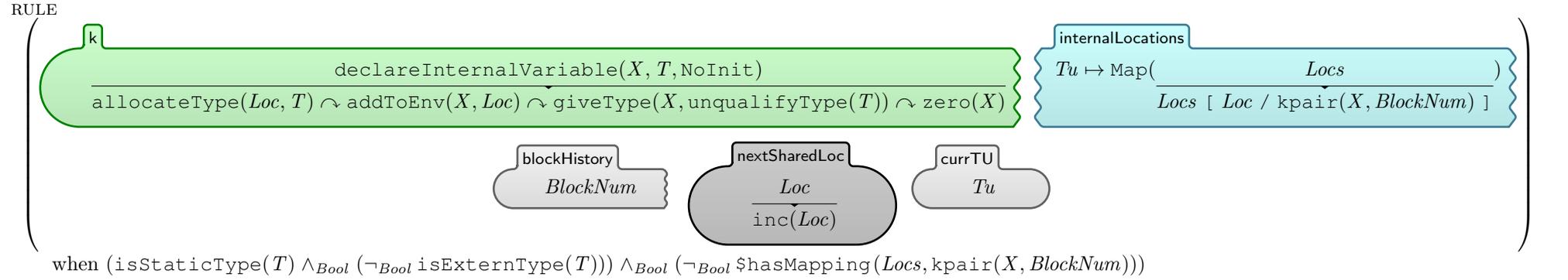
$$\frac{\text{declareInternalVariable}(X, T, \text{NoInit})}{\text{addToEnv}(X, Loc) \curvearrowright \text{giveType}(X, \text{unqualifyType}(T))}$$

internalLocations
 $Tu \mapsto \text{Map}(\text{---} (\text{kpair}(X, \text{BlockNum}) \mapsto \text{Loc}))$

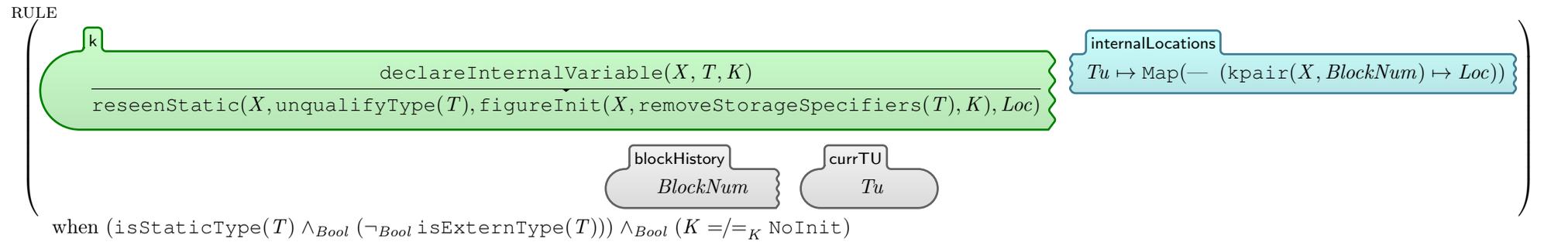
blockHistory
 BlockNum

currTU
 Tu

when $\text{isStaticType}(T) \wedge_{Bool} (\neg_{Bool} \text{isExternType}(T))$



SYNTAX $K ::= \text{reseenStatic}(Id, Type, K, Nat)$ [strict(3)]



RULE

$$\frac{\text{reseenStatic}(X, -, \text{initValue}(X, T, -), Loc)}{\text{addToEnv}(X, Loc) \curvearrowright \text{giveType}(X, T)}$$

RULE DECLAREEXTERNINTERNAL

$$\frac{\text{declareInternalVariable}(X, T, \text{NoInit})}{\text{addToEnv}(X, Loc) \curvearrowright \text{giveType}(X, \text{unqualifyType}(T))}$$

$$\frac{\text{externalLocations}}{X \mapsto Loc}$$

when $((\neg_{Bool} \text{isIncompleteType}(T)) \wedge_{Bool} (\neg_{Bool} \text{isStaticType}(T))) \wedge_{Bool} \text{isExternType}(T)$

RULE

$$\frac{\text{declareExternalVariable}(X, T, K)}{\text{declareWithLinkage}(X, T, K, \text{external})}$$

when $(\neg_{Bool} \text{isStaticType}(T)) \wedge_{Bool} (\neg_{Bool} \text{isExternType}(T))$

RULE

$$\frac{\text{declareExternalVariable}(X, T, K)}{\text{declareWithLinkage}(X, T, K, \text{external})}$$

$$\frac{\text{preLinkage}}{Tu \mapsto \text{Map}(Linkage)}$$

$$\frac{\text{currTU}}{Tu}$$

when $((\neg_{Bool} \text{isStaticType}(T)) \wedge_{Bool} \text{isExternType}(T)) \wedge_{Bool} (\neg_{Bool} \$\text{hasMapping}(Linkage, X))$

RULE DECLAREEXTERN-AGAIN

$$\frac{\text{declareExternalVariable}(X, T, K)}{\text{declareWithLinkage}(X, T, K, \text{Linkage}(X))}$$

$$\frac{\text{preLinkage}}{Tu \mapsto \text{Map}(Linkage)}$$

$$\frac{\text{currTU}}{Tu}$$

when $((\neg_{Bool} \text{isStaticType}(T)) \wedge_{Bool} \text{isExternType}(T)) \wedge_{Bool} \$\text{hasMapping}(Linkage, X)$

RULE

$$\frac{\text{declareExternalVariable}(X, T, K)}{\text{declareWithLinkage}(X, T, K, \text{internal})}$$

when $\text{isStaticType}(T) \wedge_{Bool} (\neg_{Bool} \text{isExternType}(T))$

SYNTAX $K ::= \text{declareOnly}(Id, Type, K)$
 $\quad \quad \quad | \text{declareAndDefine}(Id, Type, K, K)$

RULE

$$\frac{\text{declareWithLinkage}(X, T, \text{NoInit}, L)}{\text{declareOnly}(X, T, L)}$$

when $L \neq_K \text{noLinkage}$

$$\text{declarationOrder} \quad Tu \mapsto \text{ListToK}(\text{---} \cdot \text{---})$$

$$\frac{\cdot}{X}$$

$$\text{currTU} \quad Tu$$

RULE

$$\frac{\text{declareWithLinkage}(X, T, K, L)}{\text{declareAndDefine}(X, T, K, L)}$$

when $(K \neq_K \text{NoInit}) \wedge_{Bool} (L \neq_K \text{noLinkage})$

$$\text{declarationOrder} \quad Tu \mapsto \text{ListToK}(\text{---} \cdot \text{---})$$

$$\frac{\cdot}{X}$$

$$\text{currTU} \quad Tu$$

RULE

$$\frac{\text{declareWithLinkage}(X, T, \text{NoInit}, L)}{\text{declareOnly}(X, T, L)}$$

when $L =_K \text{noLinkage}$

$$\text{currTU} \quad Tu$$

RULE

$$\frac{\text{declareWithLinkage}(X, T, K, L)}{\text{declareAndDefine}(X, T, K, L)}$$

when $(K \neq_K \text{NoInit}) \wedge_{Bool} (L =_K \text{noLinkage})$

$$\text{currTU} \quad Tu$$

RULE

$$\frac{t(-, \text{qualifiedType}(T, \text{Extern}))}{T}$$

when isFunctionType(T)

RULE

$$\frac{\text{declareOnly}(X, T, \text{external})}{\bullet}$$

$$\frac{\bullet}{X}$$

$$Tu \mapsto \text{Map}\left(\frac{Linkage}{Linkage [\text{external} / X]}\right)$$

$$Tu \mapsto \text{Map}\left(\frac{Types}{Types [T / X]}\right)$$

$$Tu$$

when $\left((\neg_{Bool} \$hasMapping(Linkage, X)) \wedge_{Bool} (\vee_{Bool} ((Linkage(X)) ==_K \text{external})) \right) \wedge_{Bool} \left((\neg_{Bool} \$hasMapping(Types, X)) \wedge_{Bool} (\vee_{Bool} \text{isTypeCompatible}(\text{unqualifyType}(Types(X)), \text{unqualifyType}(T))) \right)$

RULE

$$\frac{\text{declareOnly}(X, t(-, \text{prototype}(T)), \text{external})}{\bullet}$$

$$Tu \mapsto \text{Map}\left(\frac{Linkage}{Linkage [\text{external} / X]}\right)$$

$$Tu \mapsto \text{Map}(Types)$$

$$Tu$$

when $\left((\neg_{Bool} \$hasMapping(Linkage, X)) \wedge_{Bool} (\vee_{Bool} ((Linkage(X)) ==_K \text{external})) \right) \wedge_{Bool} \text{isTypeCompatible}(\text{unqualifyType}(Types(X)), \text{unqualifyType}(T))$

RULE

$$\frac{\text{declareOnly}(X, T, \text{internal})}{\bullet}$$

$$\frac{\bullet}{X}$$

$$Tu \mapsto \text{Map}\left(\frac{Linkage}{Linkage [\text{internal} / X]}\right)$$

$$Tu \mapsto \text{Map}\left(\frac{Types}{Types [T / X]}\right)$$

$$Tu$$

when $(\neg_{Bool} \$hasMapping(Linkage, X)) \wedge_{Bool} (\vee_{Bool} ((Linkage(X)) ==_K \text{internal}))$

RULE

$$\frac{\text{declareOnly}(X, T, \text{noLinkage})}{\text{allocateType}(Loc, T) \rightsquigarrow \text{addToEnv}(X, Loc) \rightsquigarrow \text{giveType}(X, T)}$$

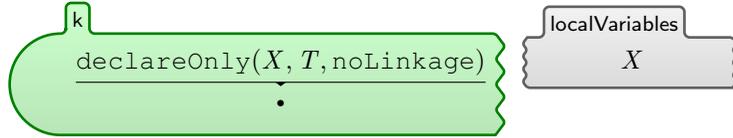
$$\frac{Loc}{\text{inc}(Loc)}$$

$$\frac{Vars \bullet}{X}$$

$$\frac{\bullet}{Loc}$$

when $((\neg_{Bool} \text{isIncompleteType}(T)) \wedge_{Bool} (\neg_{Bool} \text{isStaticType}(T))) \wedge_{Bool} (\neg_{Bool} \text{isExternType}(T)) \wedge_{Bool} (\neg_{Bool} (X \text{ in } Vars))$

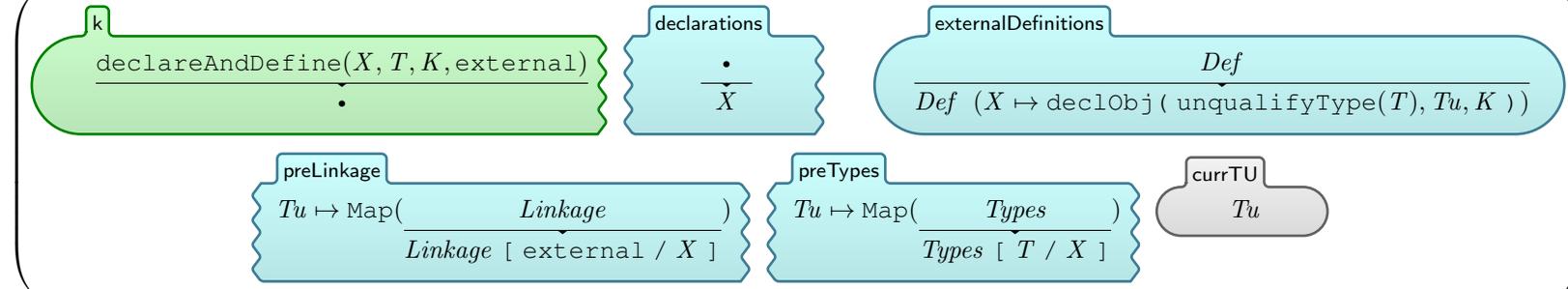
RULE



when $((\neg_{Bool} \text{isIncompleteType}(T)) \wedge_{Bool} (\neg_{Bool} \text{isStaticType}(T))) \wedge_{Bool} (\neg_{Bool} \text{isExternType}(T))$

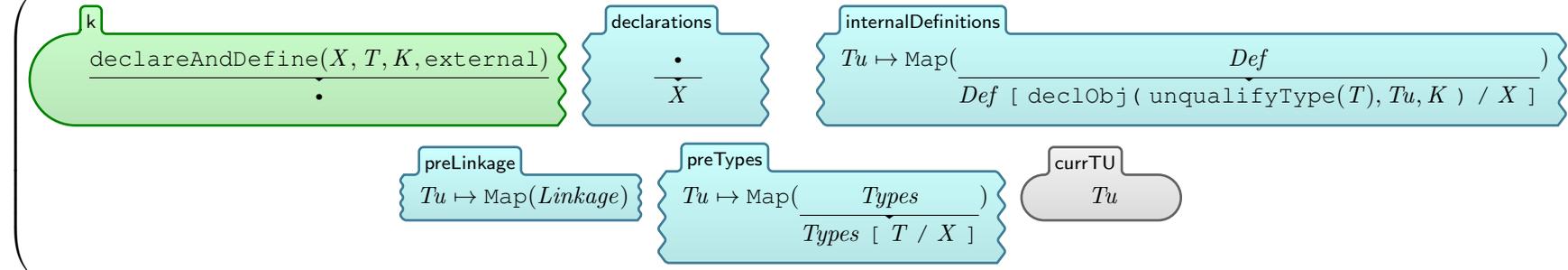
(n1570) §6.2.2 ¶4 For an identifier declared with the storage-class specifier **extern** in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

RULE

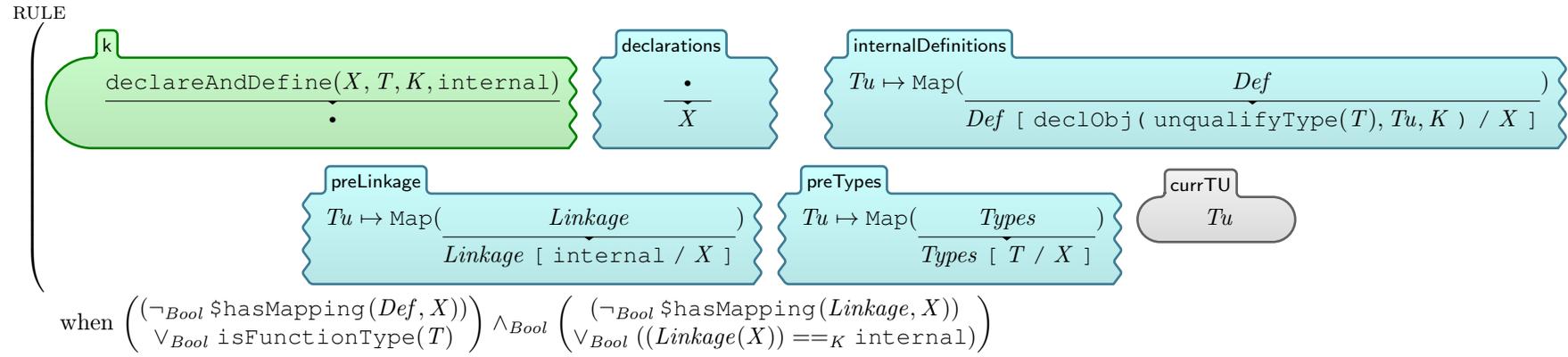


when $\left((\neg_{Bool} \$hasMapping(Def, X)) \wedge_{Bool} \left(\neg_{Bool} \$hasMapping(Linkage, X) \right) \right) \wedge_{Bool} \left(\vee_{Bool} \text{isFunctionType}(T) \right)$

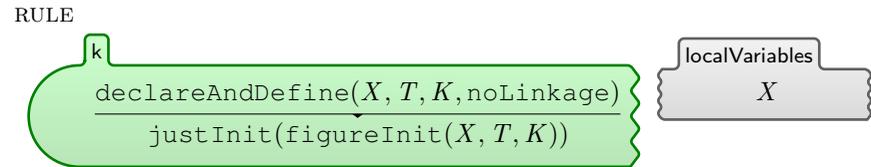
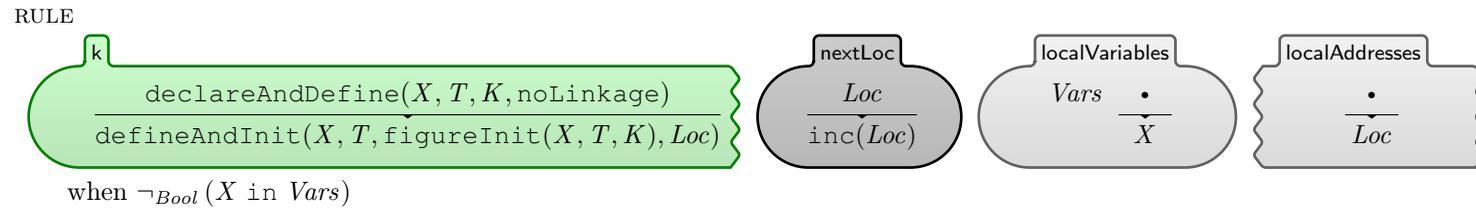
RULE



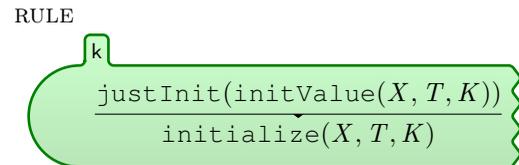
when $\left((\neg_{Bool} \$hasMapping(Def, X)) \wedge_{Bool} ((Linkage(X)) ==_K \text{internal}) \right) \wedge_{Bool} \left(\vee_{Bool} \text{isFunctionType}(T) \right)$



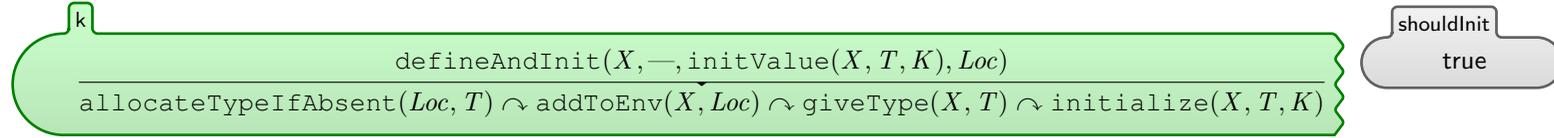
SYNTAX $K ::= \text{defineAndInit}(Id, Type, K, Nat) [\text{strict}(3)]$



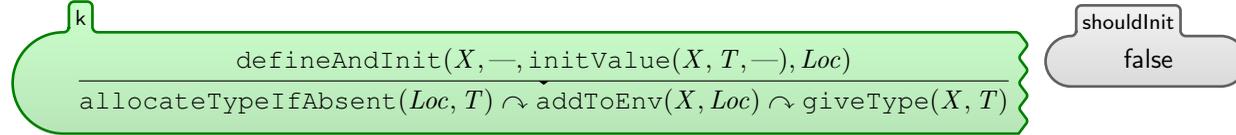
SYNTAX $K ::= \text{justInit}(K) [\text{strict}(1)]$



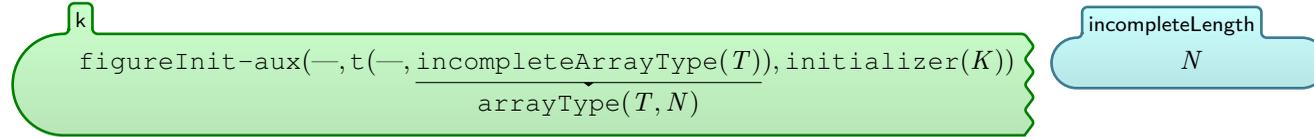
RULE



RULE

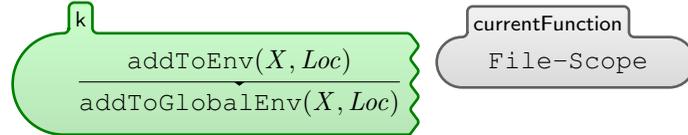


RULE

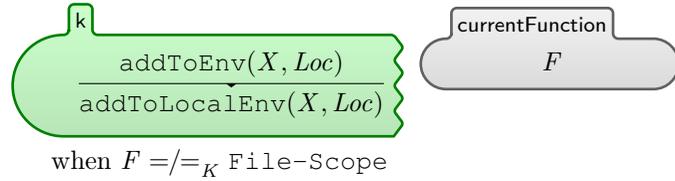


SYNTAX $K ::= \text{allocateAndZeroIfAbsent}(Type, Id)$
 $\quad | \text{addToLinkage}(Id, Type)$
 $\quad | \text{addToGlobalEnv}(K, Nat)$
 $\quad | \text{addToLocalEnv}(K, Nat)$

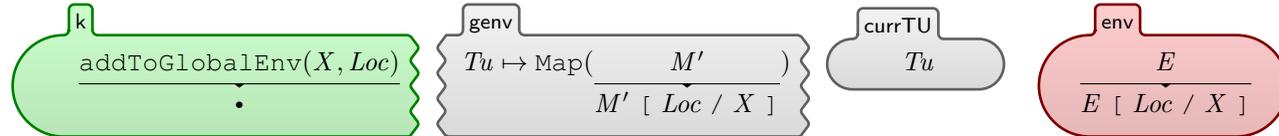
RULE



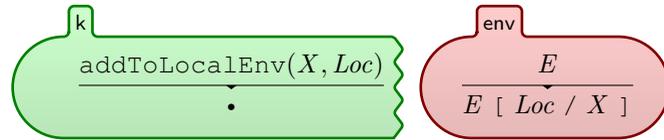
RULE



RULE

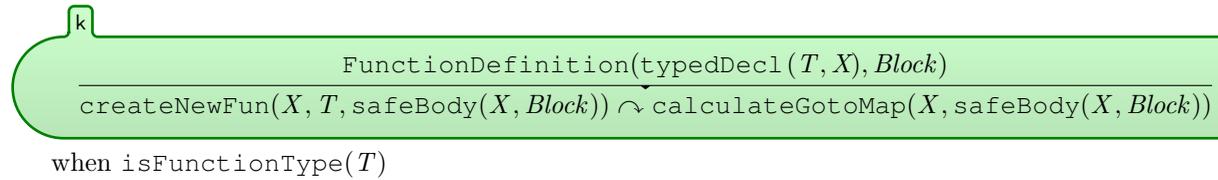


RULE



SYNTAX $K ::= \text{defineUsingOldDeclaration}(Type, Id, K)$

RULE FUNCTION-DEFINITION



SYNTAX $K ::= \text{createNewFun}(K, K, K)$

DEFINE

$$\frac{\text{createNewFun}(X, T, Block)}{\text{declareFunction}(X, T, \text{initializer}(\text{initFunction}(\& X, \text{functionObject}(X, \text{unqualifyType}(T), Block))))}$$

SYNTAX $K ::= \text{safeBody}(K, K)$

DEFINE

$$\frac{\text{safeBody}(X, Block)}{Block \rightsquigarrow \text{Return}(\text{NothingExpression})}$$

when $X \neq_K \text{Identifier}(\text{"main"})$

DEFINE

$$\frac{\text{safeBody}(\text{Identifier}(\text{"main"}), Block)}{Block \rightsquigarrow \text{Return}(0:t(\bullet, \text{int}))}$$

END MODULE

MODULE COMMON-SEMANTICS-DECLARATIONS-FUNCTION-BODY

IMPORTS COMMON-SEMANTICS-DECLARATIONS-INCLUDE

SYNTAX $K ::= \text{typingBody}(Id, Type, K)$

END MODULE

MODULE COMMON-SEMANTICS-DECLARATIONS-INITIALIZATIONS

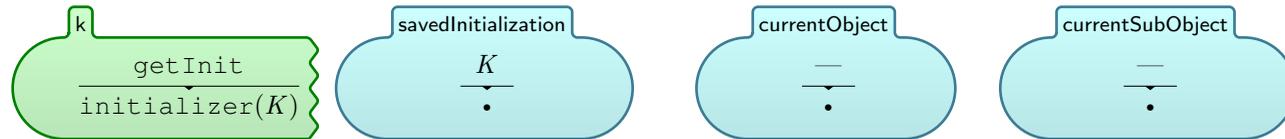
IMPORTS COMMON-SEMANTICS-DECLARATIONS-INCLUDE

SYNTAX $K ::= \text{te}(K, \text{Type})$
| getInit
| $\text{fillInit}(K)$
| $\text{fillInit-aux}(K)$
| $\text{fillInit}(\text{List}\{K\})$

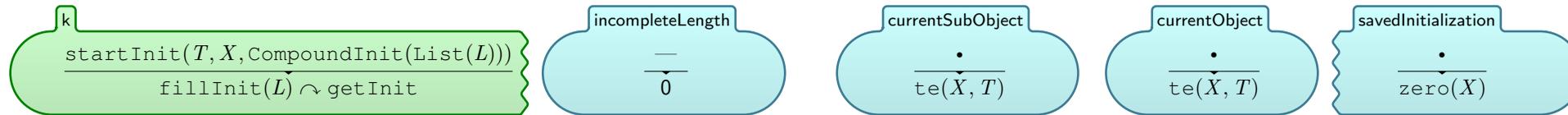
SYNTAX $C ::= \text{completeInitFragment}(K, K)$

SYNTAX $K\text{Result} ::= \text{initializerFragment}(K)$

RULE

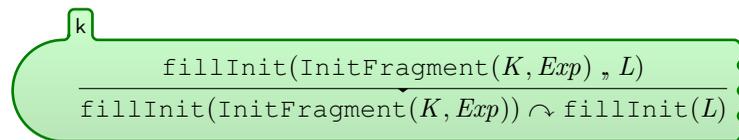


RULE



when $\text{isUnionType}(T)$
 $\vee_{\text{Bool}} \text{isAggregateType}(T)$

RULE



RULE

$$\frac{\text{fillInit}(\cdot)}{\cdot}$$

SYNTAX $ListItem ::= next$
 $| block$

RULE

$$\frac{\text{currentSubObject} \quad te(K, t(S, arrayType(T, Len)))}{te(K[0], T) \quad te(K, t(S, arrayType(T, Len)))}$$

RULE

$$\frac{\text{currentSubObject} \quad te(K, t(Se, incompleteArrayType(T)))}{te(K[0], T) \quad te(K, t(Se, incompleteArrayType(T)))}$$

RULE

$$\frac{\text{currentSubObject} \quad te(K, t(Se, structType(S)))}{te(K.F, T) \quad te(K, t(Se, structType(S)))} \quad \text{structs} \quad S \mapsto \text{aggregateInfo}(\text{typedDecl}(T, F), -, -, -)$$

RULE

$$\frac{\text{currentSubObject} \quad te(K, t(Se, unionType(S)))}{te(K.F, T) \quad te(K, t(Se, unionType(S)))} \quad \text{structs} \quad S \mapsto \text{aggregateInfo}(\text{typedDecl}(T, F), -, -, -)$$

RULE INIT-NEXT-ARRAY-ELEMENT

$$\frac{\text{currentSubObject} \quad next \quad te(K[N], T) \quad te(K, t(-, arrayType(-, Len)))}{te(K[N +_{Int} 1], T)}$$

when $Len >_{Int} (N +_{Int} 1)$

RULE INIT-NEXT-ARRAY-ELEMENT-DONE

$$\frac{\text{currentSubObject} \quad \text{next } \frac{\text{te}(K[N], T) \quad \text{te}(K, t(-, \text{arrayType}(-, Len)))}{\cdot}}{\text{when } \neg_{Bool} (Len >_{Int} (N +_{Int} 1))}$$

RULE INIT-NEXT-INCOMPLETE-ARRAY-ELEMENT

$$\frac{\text{currentSubObject} \quad \text{next } \frac{\text{te}(K[N], T) \quad \text{te}(K, t(-, \text{incompleteArrayType}(-)))}{\text{te}(K[N +_{Int} 1], T)}}{\text{te}(K[N +_{Int} 1], T)}$$

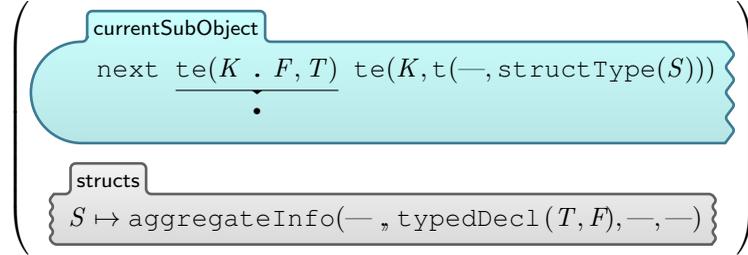
RULE INIT-NEXT-STRUCT-ELEMENT

$$\left(\frac{\text{currentSubObject} \quad \text{next } \frac{\text{te}(K \cdot F, T) \quad \text{te}(K, t(Se, \text{structType}(S)))}{\text{te}(K \cdot F', T') \quad \text{te}(K, t(Se, \text{structType}(S)))}}{\text{structs} \quad \{ S \mapsto \text{aggregateInfo}(-, \text{typedDecl}(T, F), \text{typedDecl}(T', F'), -, -, -) \}}}{\text{when } F' \neq_K \#NoName}$$

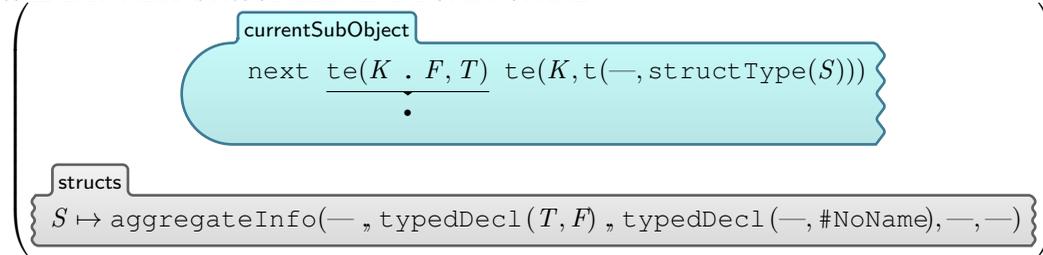
RULE INIT-NEXT-STRUCT-ELEMENT-NONAME

$$\left(\frac{\text{currentSubObject} \quad \text{next } \frac{\text{te}(K \cdot F, T) \quad \text{te}(K, t(Se, \text{structType}(S)))}{\text{te}(K \cdot F', T') \quad \text{te}(K, t(Se, \text{structType}(S)))}}{\text{structs} \quad \{ S \mapsto \text{aggregateInfo}(-, \text{typedDecl}(T, F), \text{typedDecl}(-, \#NoName), \text{typedDecl}(T', F'), -, -, -) \}})$$

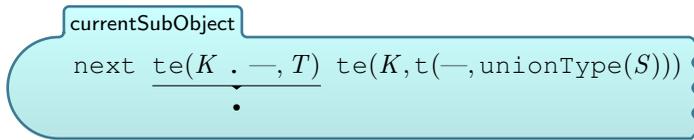
RULE INIT-NEXT-STRUCT-ELEMENT-DONE



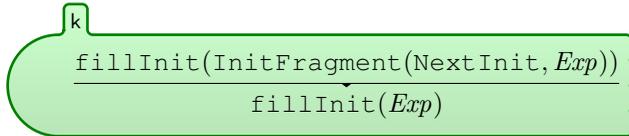
RULE INIT-NEXT-STRUCT-ELEMENT-DONE-NONAME



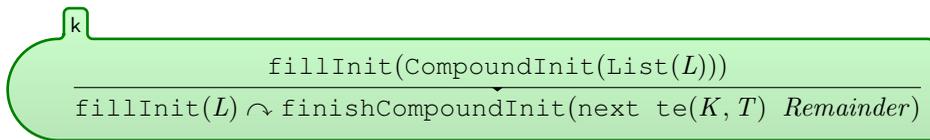
RULE INIT-NEXT-UNION-ELEMENT-DONE



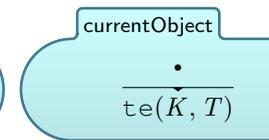
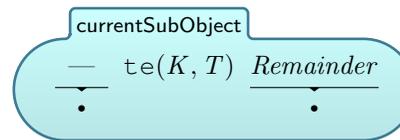
RULE

SYNTAX $K ::= \text{finishCompoundInit}(\text{List})$

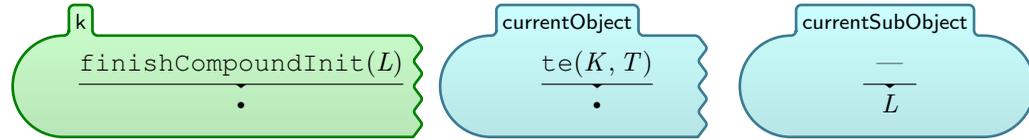
RULE



when $\text{isAggregateType}(T)$
 $\vee_{Bool} \text{isUnionType}(T)$



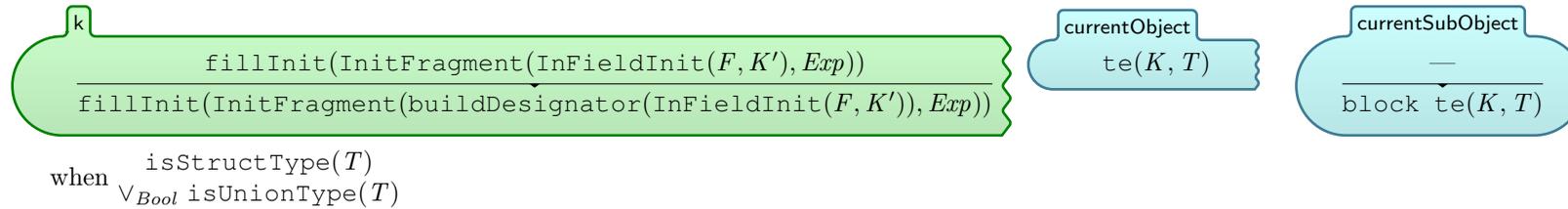
RULE



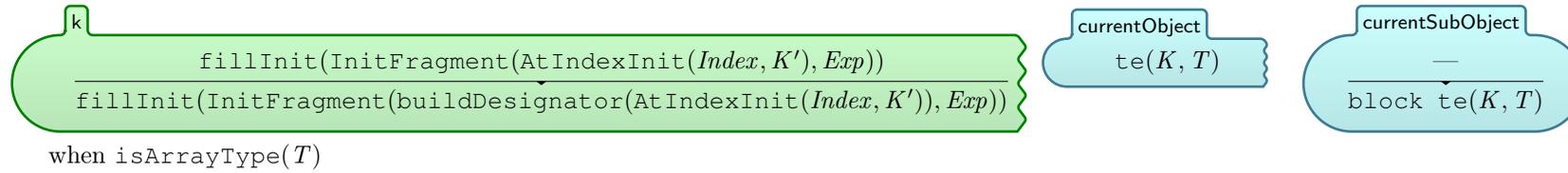
CONTEXT: fillInit(InitFragment(\square , —))

SYNTAX $K ::= \text{buildDesignator}(K)$

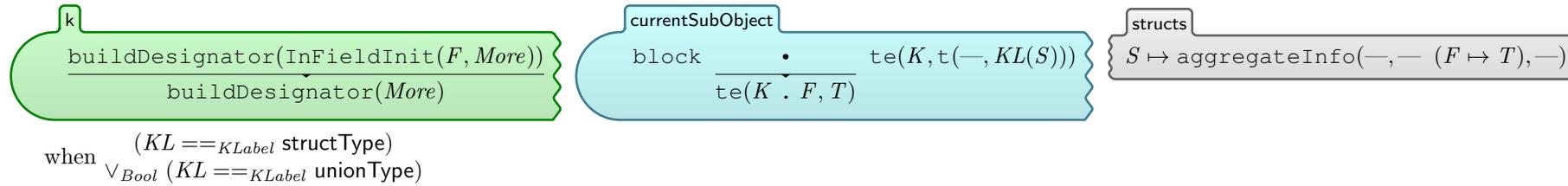
RULE



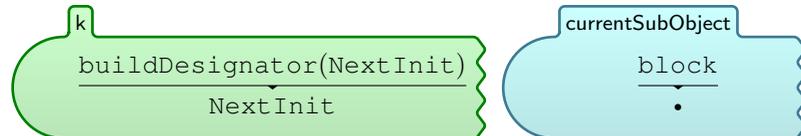
RULE



RULE



RULE



DEFINE **INNERTYPE-ARRAYTYPE**

$$\frac{\text{innerType}(t(-, \text{arrayType}(T, -)))}{T}$$

DEFINE **INNERTYPE-INCOMPLETEARRAYTYPE**

$$\frac{\text{innerType}(t(-, \text{incompleteArrayType}(T)))}{T}$$

DEFINE **INNERTYPE-FLEXIBLEARRAYTYPE**

$$\frac{\text{innerType}(t(-, \text{flexibleArrayType}(T)))}{T}$$

DEFINE **INNERTYPE-QUALIFIEDTYPE**

$$\frac{\text{innerType}(t(-, \text{qualifiedType}(T, -)))}{\text{innerType}(T)}$$

DEFINE **INNERTYPE-POINTERTYPE**

$$\frac{\text{innerType}(t(-, \text{pointerType}(T)))}{T}$$

DEFINE **INNERTYPE-BITFIELDTYPE**

$$\frac{\text{innerType}(t(-, \text{bitFieldType}(T, -)))}{T}$$

DEFINE **INNERTYPE-FUNCTIONTYPE**

$$\frac{\text{innerType}(t(-, \text{functionType}(T, -)))}{T}$$

CONTEXT: $\text{buildDesignator}(\text{AtIndexInit}(\frac{\square}{\text{reval}(\square)}, -))$

RULE

$\frac{\text{buildDesignator}(\text{AtIndexInit}(N:-, \text{More}))}{\text{buildDesignator}(\text{More})}$	$\text{block} \frac{\text{currentSubObject} \quad \bullet \quad \text{te}(K, T)}{\text{te}(K[N], \text{innerType}(T))}$
when isArrayType(T)	

SYNTAX $K ::= \text{popInit}$

SYNTAX $\text{Nat} ::= \text{getTopArrayUse}(K) [\text{function}]$

DEFINE
 $\frac{\text{getTopArrayUse}(X)}{0}$

DEFINE
 $\frac{\text{getTopArrayUse}(X[N])}{N + \text{Int } 1}$

DEFINE
 $\frac{\text{getTopArrayUse}(K . F)}{\text{getTopArrayUse}(K)}$

DEFINE
 $\frac{\text{getTopArrayUse}((K[N])[-])}{\text{getTopArrayUse}(K[N])}$

DEFINE
 $\frac{\text{getTopArrayUse}((K . F)[-])}{\text{getTopArrayUse}(K)}$

SYNTAX $K ::= \text{initializeSingleInit}(K)$

RULE

$\frac{\text{•} \quad \text{initializeSingleInit}(K)}{\text{typeof}(K)}$

RULE

$\frac{\text{•} \quad T' \sim \text{initializeSingleInit}(K')}{\text{•}}$ $\frac{\text{•} \quad \text{te}(K, T)}{\text{next}}$ $\frac{\text{incompleteLength} \quad N}{\text{maxInt}(N, \text{getTopArrayUse}(K))}$ $\frac{\text{savedInitialization} \quad \text{•}}{\text{AllowWrite}(K) := K'; \sim \text{possiblyMakeConst}(T, K)}$

when $\left(\left(\left(\begin{array}{c} \text{isBasicType}(T) \\ \vee_{Bool} \text{isPointerType}(T) \\ \vee_{Bool} \text{isBitfieldType}(T) \end{array} \right) \wedge_{Bool} (\neg_{Bool} \text{isStructType}(T')) \right) \wedge_{Bool} (\neg_{Bool} \text{isUnionType}(T')) \right)$

SYNTAX $K ::= \text{possiblyMakeConst}(Type, K)$

RULE

$$\frac{\text{possiblyMakeConst}(T, K)}{\text{makeUnwritableSubObject}(K)}$$

when $\text{isConstType}(T)$

RULE

$$\frac{\text{possiblyMakeConst}(T, K)}{\cdot}$$

when $\neg_{Bool} \text{isConstType}(T)$

SYNTAX $K ::= \text{initFromAggregateRHS}(K, Type)$

RULE

$$\frac{T \curvearrowright \text{initializeSingleInit}(K)}{\text{initFromAggregateRHS}(K, T)}$$

when $\text{isStructType}(T)$
 $\vee_{Bool} \text{isUnionType}(T)$

SYNTAX $K ::= \text{initFromStructRHS}(K, Type)$

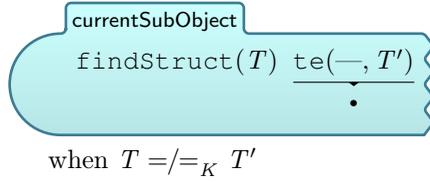
RULE

$$\frac{\text{initFromAggregateRHS}(K, t(S, \text{structType}(S)))}{\text{initFromStructRHS}(K, t(S, \text{structType}(S)))}$$

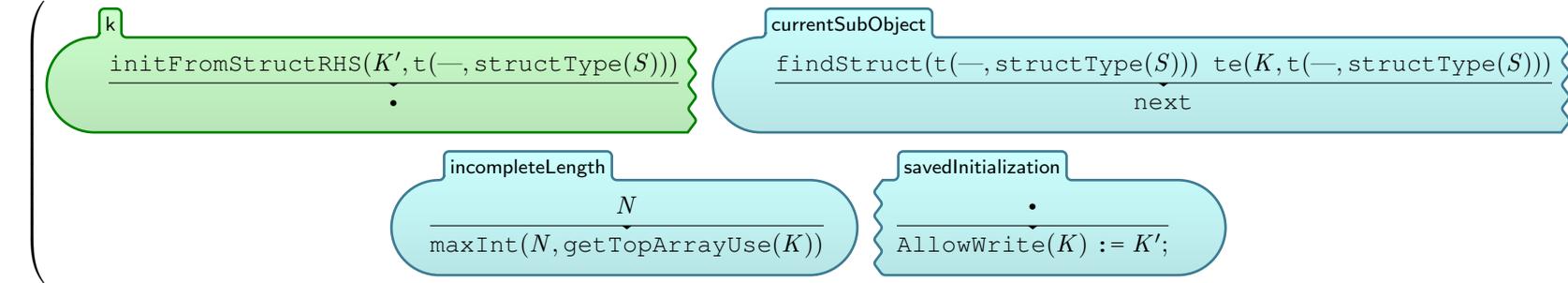
$$\frac{\text{currentSubObject}}{\text{findStruct}(t(S, \text{structType}(S)))}$$

SYNTAX $ListItem ::= \text{findStruct}(Type)$

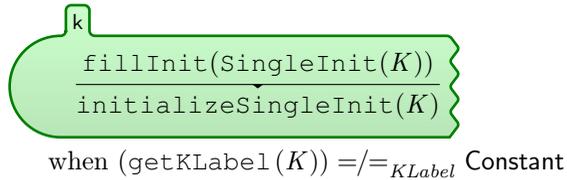
RULE



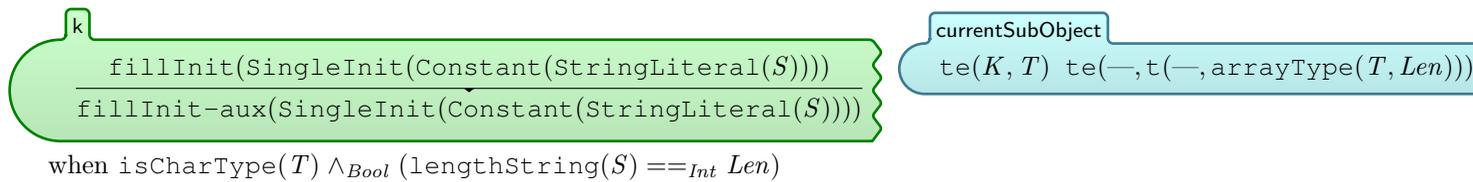
RULE



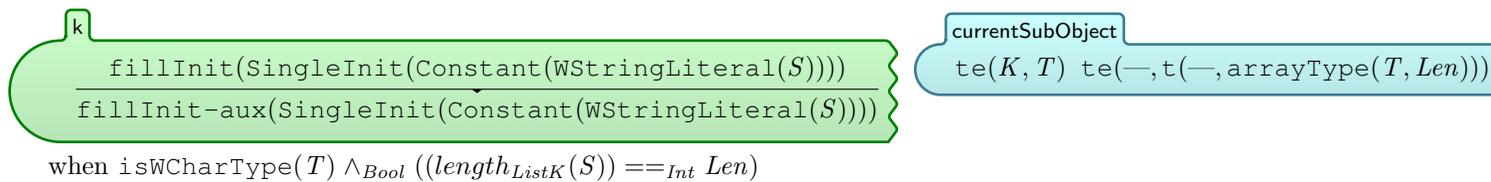
RULE



RULE FILLINIT-STRING-ARRAY-EQ



RULE FILLINIT-WSTRING-ARRAY-EQ



RULE FILLINIT-STRING-ARRAY-LT

$$\frac{\text{k} \quad \frac{\text{fillInit}(\text{SingleInit}(\text{Constant}(\text{StringLiteral}(S))))}{\text{fillInit}(\text{SingleInit}(\text{Constant}(\text{StringLiteral}(S +_{\text{String}} \text{"\000"})))}}{\text{when } \text{isCharType}(T) \wedge_{\text{Bool}} (\text{lengthString}(S) <_{\text{Int}} \text{Len})} \quad \begin{array}{l} \text{currentSubObject} \\ \text{te}(K, T) \text{ te}(-, \text{t}(-, \text{arrayType}(T, \text{Len}))) \end{array}}$$

RULE FILLINIT-WSTRING-ARRAY-LT

$$\frac{\text{k} \quad \frac{\text{fillInit}(\text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S))))}{\text{fillInit}(\text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S, 0))))}}{\text{when } \text{isWCharType}(T) \wedge_{\text{Bool}} ((\text{length}_{\text{List}K}(S)) <_{\text{Int}} \text{Len})} \quad \begin{array}{l} \text{currentSubObject} \\ \text{te}(K, T) \text{ te}(-, \text{t}(-, \text{arrayType}(T, \text{Len}))) \end{array}}$$

RULE FILLINIT-STRING-CHAR

$$\frac{\text{k} \quad \frac{\text{fillInit}(\text{SingleInit}(\text{Constant}(\text{StringLiteral}(S))))}{\text{fillInit-aux}(\text{SingleInit}(\text{Constant}(\text{StringLiteral}(S +_{\text{String}} \text{"\000"})))}}{\text{when } \text{isCharType}(T)} \quad \begin{array}{l} \text{currentSubObject} \\ \text{te}(K, T) \text{ te}(-, \text{t}(-, \text{incompleteArrayType}(T))) \end{array}}$$

RULE FILLINIT-WSTRING-WCHAR

$$\frac{\text{k} \quad \frac{\text{fillInit}(\text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S))))}{\text{fillInit-aux}(\text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S, 0))))}}{\text{when } \text{isWCharType}(T)} \quad \begin{array}{l} \text{currentSubObject} \\ \text{te}(K, T) \text{ te}(-, \text{t}(-, \text{incompleteArrayType}(T))) \end{array}}$$

RULE FILLINIT-AUX-STRING-SOME

$$\frac{\text{k} \quad \frac{\text{fillInit-aux}(\text{SingleInit}(\text{Constant}(\text{StringLiteral}(S))))}{\text{fillInit}(\text{initHead}(S, T)) \curvearrowright \text{fillInit-aux}(\text{initTail}(S))}}{\text{when } (S \neq_{\text{String}} \text{""}) \wedge_{\text{Bool}} \text{isCharType}(T)} \quad \begin{array}{l} \text{currentSubObject} \\ \text{te}(K, T) \end{array}}$$

SYNTAX $K ::= \text{initHead}(K, K)$
 $\quad \quad \quad | \text{initTail}(K)$

DEFINE

$\text{initHead}(S, T)$

$\text{SingleInit}(\text{charToAscii}(\text{firstChar}(S)) : \text{t}(\text{getModifiers}(T), \text{char}))$

DEFINE

$\text{initTail}(S)$

$\text{SingleInit}(\text{Constant}(\text{StringLiteral}(\text{butFirstChar}(S))))$

RULE **FILLINITAUX-WSTRING-SOME**

k

$\frac{\text{fillInit-aux}(\text{SingleInit}(\text{Constant}(\text{WStringLiteral}(N, S))))}{\text{fillInit}(\text{SingleInit}(N : \text{cfg:wcharut})) \rightsquigarrow \text{fillInit-aux}(\text{SingleInit}(\text{Constant}(\text{WStringLiteral}(S))))}$

currentSubObject
 $\text{te}(K, T)$

when $\text{isWCharType}(T)$

RULE **FILLINITAUX-STRING-DONE**

k

$\text{fillInit-aux}(\text{SingleInit}(\text{Constant}(\text{StringLiteral}(""))))$

•

RULE **FILLINITAUX-WSTRING-DONE**

k

$\text{fillInit-aux}(\text{SingleInit}(\text{Constant}(\text{WStringLiteral}(\cdot))))$

•

RULE **FILLINIT-STRING-NOTCHAR**

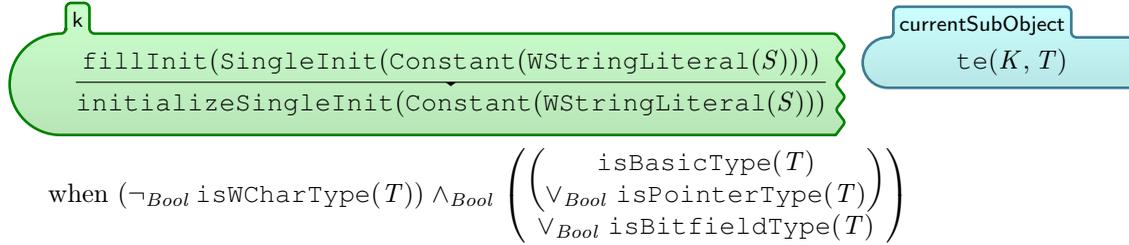
k

$\frac{\text{fillInit}(\text{SingleInit}(\text{Constant}(\text{StringLiteral}(S))))}{\text{initializeSingleInit}(\text{Constant}(\text{StringLiteral}(S)))}$

currentSubObject
 $\text{te}(K, T)$

when $(\neg_{Bool} \text{isCharType}(T)) \wedge_{Bool} \left(\begin{array}{l} \text{isBasicType}(T) \\ \vee_{Bool} \text{isPointerType}(T) \\ \vee_{Bool} \text{isBitFieldType}(T) \end{array} \right)$

RULE **FILLINIT-WSTRING-NOTWCHAR**



END MODULE

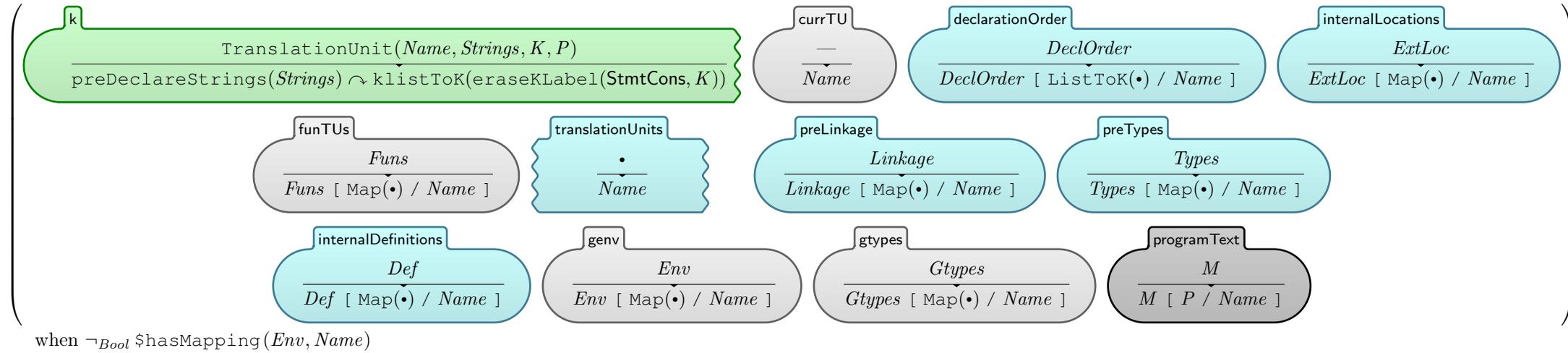
MODULE COMMON-SEMANTICS-DECLARATIONS-RESOLUTION

IMPORTS COMMON-SEMANTICS-DECLARATIONS-INCLUDE

SYNTAX $K ::= \text{canonicalizeTranslationUnitVariables}$

RULE **UNPACK-TRANSLATIONUNIT**

270



SYNTAX $K ::= \text{preDeclareStrings}(K)$

RULE

$$\frac{\text{preDeclareStrings}(\text{List}(K, L))}{K \rightsquigarrow \text{discard} \rightsquigarrow \text{preDeclareStrings}(\text{List}(L))}$$

RULE

$$\frac{\text{preDeclareStrings}(\text{List}(\cdot))}{\cdot}$$

SYNTAX $K ::= \text{resolve}(K)$

RULE

$$\frac{\cdot \rightsquigarrow \text{resolveReferences}}{\text{resolve}(Tu)}$$

$$\frac{Tu}{\cdot}$$

SYNTAX $K ::= \text{resolveLeftovers}$

RULE

$$\frac{\text{resolveReferences}}{\text{resolveLeftovers}}$$

$$\frac{\cdot}{\text{translationUnits}}$$

RULE

$$\frac{\cdot \rightsquigarrow \text{resolveLeftovers}}{\text{addToEnv}(X, Loc) \rightsquigarrow \text{giveType}(X, \text{unqualifyType}(T))}$$

$$\frac{-}{Tu}$$

$$X \mapsto Loc$$

$$\frac{\text{leftover}(Tu, X, T)}{\cdot}$$

RULE

$$\frac{\text{resolveLeftovers}}{\cdot}$$

$$Locs$$

$$\frac{\text{leftover}(-, X, -)}{\cdot}$$

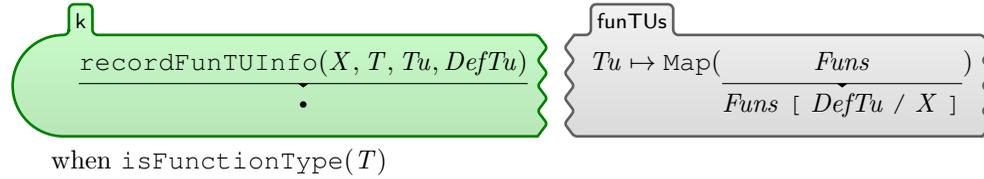
when $\neg_{Bool} \$hasMapping(Locs, X)$

RULE

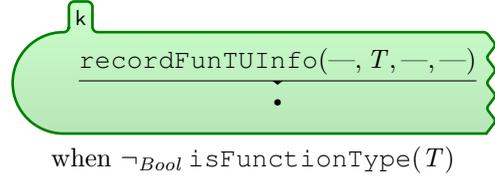


SYNTAX $K ::= \text{resolveInternal}(Id, K, K)$
 $\quad | \text{resolveExternal}(Id, K, Bag, K)$
 $\quad | \text{resolveExternal}'(Id, K, Bag, K, Nat, K)$ [strict(6)]
 $\quad | \text{recordFunTUInfo}(K, Type, K, K)$

RULE

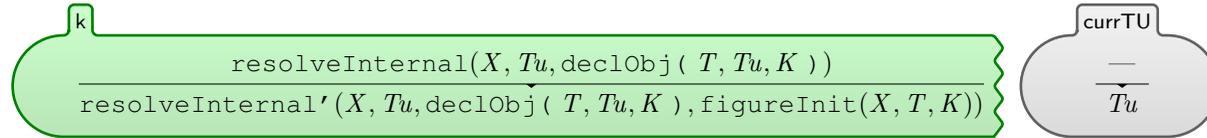


RULE

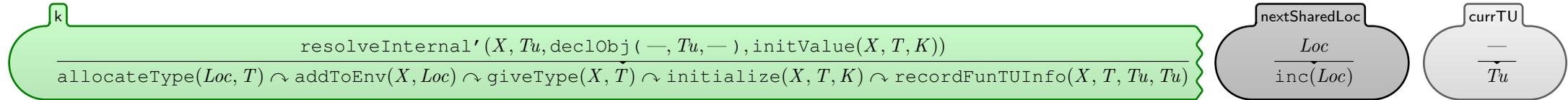


SYNTAX $K ::= \text{allocateWithInit}(K, Nat)$ [strict]
 $\quad | \text{noAllocateWithInit}(K, Nat)$ [strict]
 $\quad | \text{resolveInternal}'(Id, K, K, K)$ [strict(4)]

RULE



RULE



RULE

$$\frac{\text{resolveExternal}(X, Tu, Units, \text{declObj}(T, DefTu, K))}{\text{resolveExternal}'(X, Tu, Units, \text{declObj}(T, DefTu, K), Loc, \text{figureInit}(X, T, K))} \quad \text{nextSharedLoc} \frac{Loc}{\text{inc}(Loc)}$$

RULE

$$\frac{\text{addToEnv}(X, Loc) \curvearrow \text{giveType}(X, T) \curvearrow \text{recordFunTUInfo}(X, T, Tu', DefTu) \curvearrow \text{resolveExternal}'(X, Tu, Tu', Units, \text{declObj}(_, DefTu, _), Loc, \text{initValue}(X, T, K))}{\bullet} \quad \text{currTU} \frac{_}{Tu'} \quad \text{preLinkage} \frac{Tu' \mapsto \text{Map}(_ X \mapsto \text{external})}{\bullet}$$

RULE

$$\frac{\text{resolveExternal}'(X, Tu, \bullet, \text{declObj}(_, DefTu, _), Loc, \text{initValue}(X, T, K))}{\text{allocateType}(Loc, T) \curvearrow \text{addToEnv}(X, Loc) \curvearrow \text{giveType}(X, T) \curvearrow \text{initialize}(X, T, K) \curvearrow \text{recordFunTUInfo}(X, T, Tu, DefTu)} \quad \text{currTU} \frac{_}{Tu} \quad \text{externalLocations} \frac{Locs}{Locs [Loc / X]}$$

when $\neg_{Bool} \$hasMapping(Locs, X)$

RULE

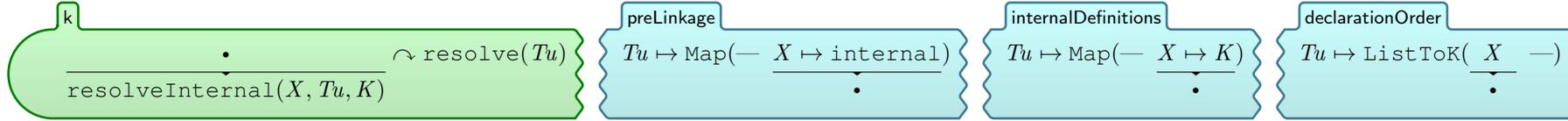
$$\frac{\text{resolveExternal}'(X, _, Tu, Units, _, _, _)}{\bullet} \quad \text{preLinkage} \frac{Tu \mapsto \text{Map}(M)}{\bullet}$$

when $\neg_{Bool} \$hasMapping(M, X)$

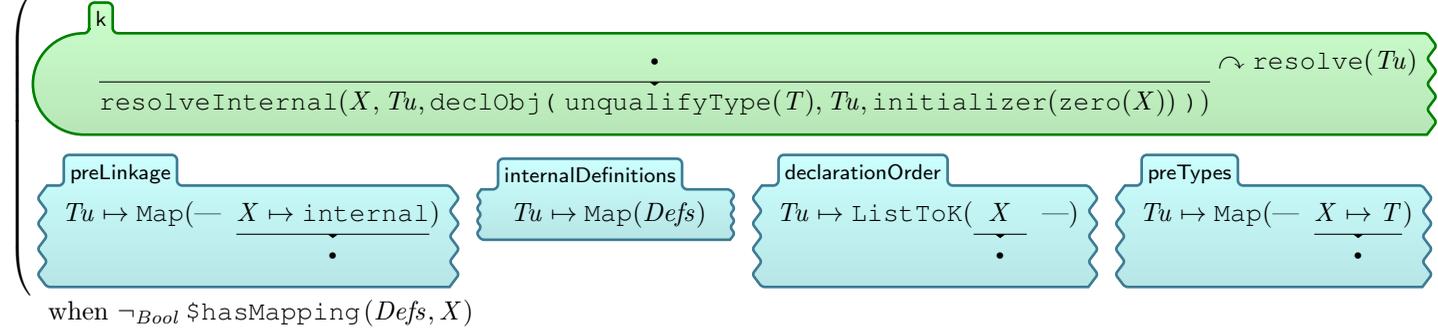
RULE RESOLVEEXTERNAL-INTERNAL

$$\frac{\text{resolveExternal}'(X, _, Tu, Units, _, _, _)}{\bullet} \quad \text{preLinkage} \frac{Tu \mapsto \text{Map}(_ (X \mapsto \text{internal}))}{\bullet}$$

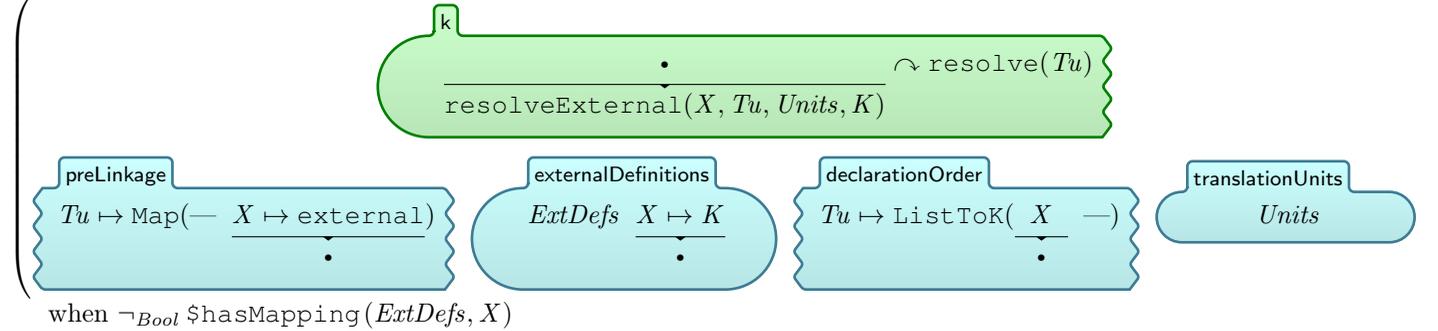
RULE



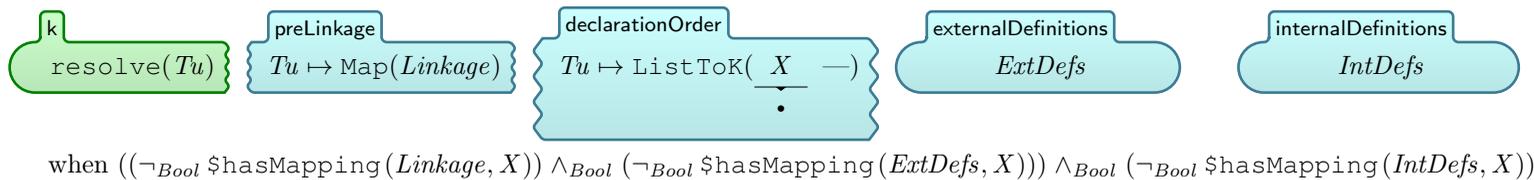
RULE



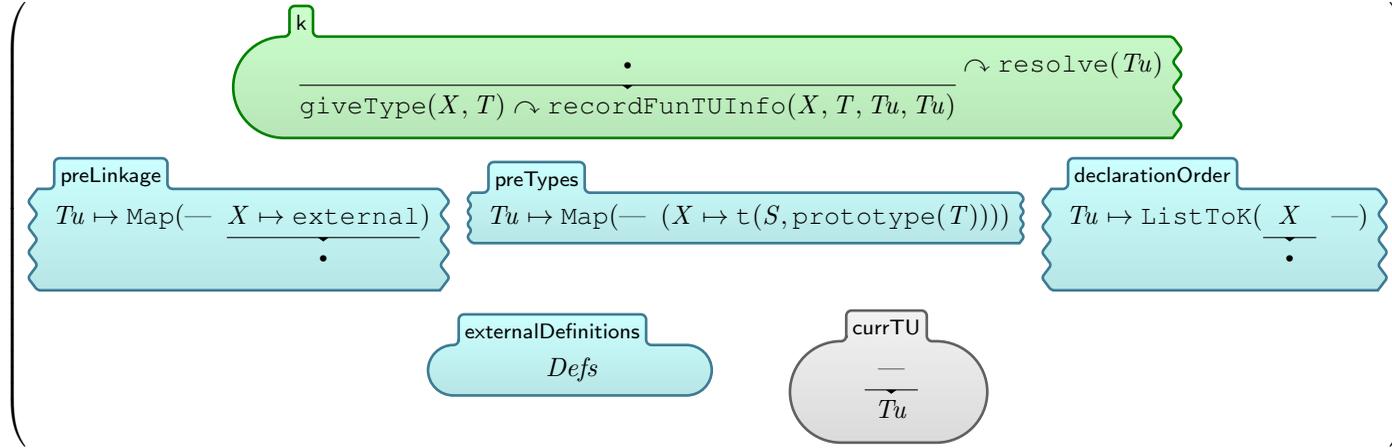
RULE



RULE

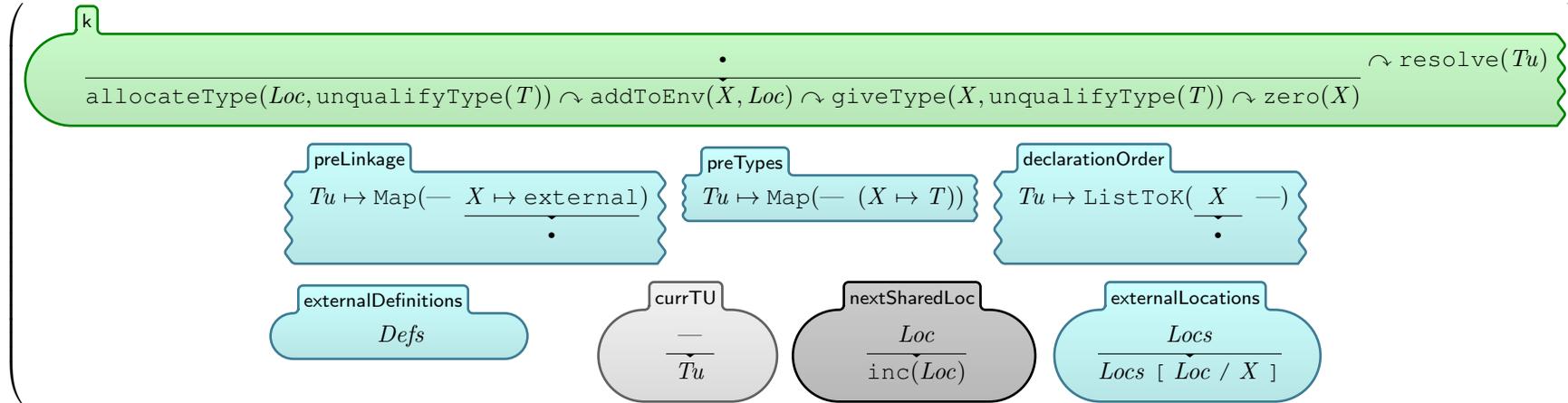


RULE



when $\text{isFunctionType}(T) \wedge_{Bool} (\neg_{Bool} \$\text{hasMapping}(Defs, X))$

RULE

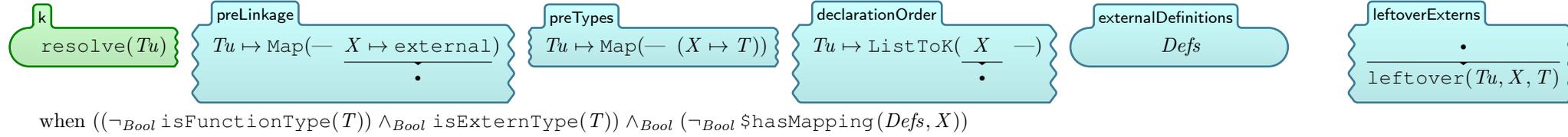


when $((((\neg_{Bool} \$\text{hasMapping}(Locs, X)) \wedge_{Bool} (\neg_{Bool} \text{isFunctionType}(T))) \wedge_{Bool} (\neg_{Bool} \text{isIncompleteType}(T))) \wedge_{Bool} (\neg_{Bool} \text{isExternType}(T)))$
 $(\neg_{Bool} \$\text{hasMapping}(Defs, X))$

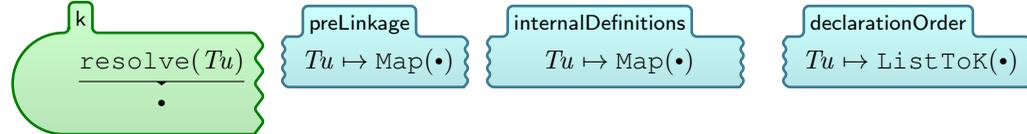
\wedge_{Bool}

SYNTAX $K ::= \text{leftover}(K, K, K)$

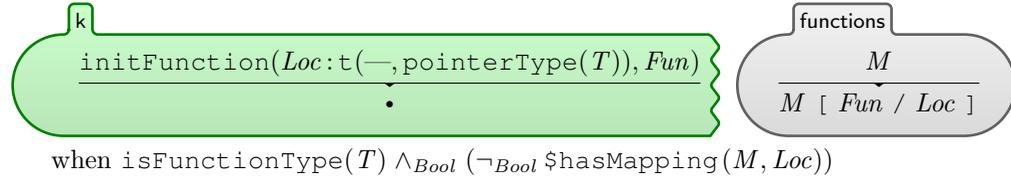
RULE **RESOLVE-EXTERN-OBJECT-NOMAPPING**



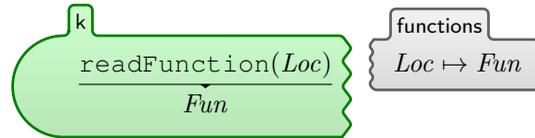
RULE



RULE **INITIALIZE-FUNCTION**



RULE



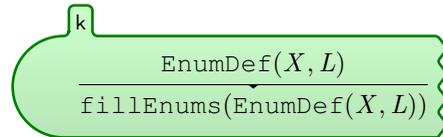
END MODULE

MODULE **COMMON-SEMANTICS-DECLARATIONS-ENUMS**

IMPORTS **COMMON-SEMANTICS-DECLARATIONS-INCLUDE**

SYNTAX $K ::= \text{fillEnums}(K)$
 $\quad \quad \quad | \text{fillEnums-aux}(K, K)$

RULE



RULE

$$\frac{\text{fillEnums}(K)}{\text{fillEnums-aux}(K, 0:t(\bullet, \text{int}))}$$

RULE

$$\frac{\text{fillEnums-aux}(\text{EnumDef}(X, \text{List}(\text{EnumItem}(E), L)), K)}{\text{doDeclare}(\text{typedDecl}(t(\bullet, \text{int}), E), \text{SingleInit}(K)) \rightsquigarrow \text{fillEnums-aux}(\text{EnumDef}(X, \text{List}(L)), K + 1:t(\bullet, \text{int}))}$$

RULE

$$\frac{\text{fillEnums-aux}(\text{EnumDef}(X, \text{List}(\text{EnumItemInit}(E, \text{Exp}), L)), -)}{\text{doDeclare}(\text{typedDecl}(t(\bullet, \text{int}), E), \text{SingleInit}(\text{Exp})) \rightsquigarrow \text{fillEnums-aux}(\text{EnumDef}(X, \text{List}(L)), \text{Exp} + 1:t(\bullet, \text{int}))}$$

when $\text{Exp} \neq_K \text{NothingExpression}$

RULE

$$\frac{\text{fillEnums-aux}(\text{EnumDef}(X, \text{List}(\bullet)), -)}{\bullet}$$

END MODULE

MODULE COMMON-C-DECLARATIONS

IMPORTS COMMON-SEMANTICS-DECLARATIONS-INCLUDE

IMPORTS COMMON-SEMANTICS-DECLARATIONS-GENERAL

IMPORTS COMMON-SEMANTICS-DECLARATIONS-FUNCTION-BODY

IMPORTS COMMON-SEMANTICS-DECLARATIONS-INITIALIZATIONS

IMPORTS COMMON-SEMANTICS-DECLARATIONS-ENUMS

IMPORTS COMMON-SEMANTICS-DECLARATIONS-RESOLUTION

END MODULE

MODULE DYNAMIC-SEMANTICS-DECLARATIONS-INCLUDE

IMPORTS DYNAMIC-INCLUDE

IMPORTS COMMON-SEMANTICS-DECLARATIONS-INCLUDE

END MODULE

MODULE DYNAMIC-SEMANTICS-DECLARATIONS-BINDING

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-INCLUDE

SYNTAX $K ::= \text{bind-aux}(\text{Nat}, \text{List}\{K\text{Result}\}, \text{List}\{K\text{Result}\})$

RULE

$$\frac{\text{bind}(L, L')}{\text{bind-aux}(\text{NullPointer}, L, L')}$$

RULE BIND-EMPTY-VOID

$$\frac{\text{bind-aux}(-, \bullet, \text{typedDecl}(t(\bullet, \text{void}), -))}{\text{sequencePoint}}$$

RULE BIND-EMPTY

$$\frac{\text{bind-aux}(-, \bullet, \bullet)}{\text{sequencePoint}}$$

RULE BIND-COERCE-ARRAY

$$\text{bind-aux}(-, L, - , \text{typedDecl}(t(-, \text{arrayType}(T, -)), X) , -)$$

$$\frac{\text{pointerType}(T)}$$

[anywhere]

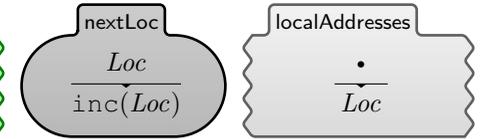
RULE **BIND-COERCE-INCOMPLETEARRAY**

$$\text{bind-aux}(_, L, _, \text{typedDecl}(\text{t}(_, \text{incompleteArrayType}(T)), X), _) \xrightarrow{\text{pointerType}(T)}$$

[anywhere]

RULE **BIND-ONE**

$$\frac{\text{true} \curvearrow \text{bind-aux}(_, V : T', _, L, \text{typedDecl}(T, X), _) \quad \text{allocateType}(Loc, T) \curvearrow \text{addToEnv}(X, Loc) \curvearrow \text{giveType}(X, T) \curvearrow \text{initialize}(X, T, \text{AllowWrite}(X) := V : T';) \curvearrow \text{bind-aux}(Loc, L, P)}{\text{when } \neg_{Bool} \text{ isArrayType}(T)}$$



RULE **BIND-ONE-CHECK-TYPE**

$$\frac{\bullet \curvearrow \text{bind-aux}(_, V : T', _, L, \text{typedDecl}(T, X), _) \quad \text{isTypeCompatible}(T, T'){\cdot}}$$

279

SYNTAX $List\{K\} ::= \text{promoteList}(List\{KResult\})$ [function]

DEFINE **PROMOTELIST-NEEDS-PROMOTING**

$$\frac{\text{promoteList}(V : \text{t}(S, T), _) \quad \text{cast}(\text{argPromote}(\text{t}(S, T)), V : \text{t}(S, T), _, \text{promoteList}(L))}{\text{when } \left(\left(\left(\text{rank}(\text{t}(S, T)) <_{Int} \text{rank}(\text{t}(\bullet, \text{int})) \right) \wedge_{Bool} \text{hasIntegerType}(\text{t}(S, T)) \right) \vee_{Bool} \text{isBitFieldType}(\text{t}(S, T)) \right) \wedge_{Bool} (T ==_K \text{float})}$$

DEFINE **PROMOTELIST-PROMOTED**

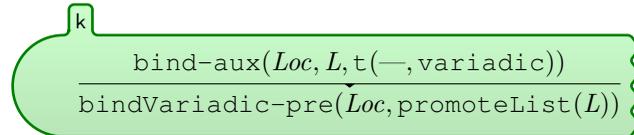
$$\frac{\text{promoteList}(V : \text{t}(S, T), _) \quad V : \text{t}(S, T), _, \text{promoteList}(L)}{\text{when } \left(\left(\left(\left(\neg_{Bool} \text{hasIntegerType}(\text{t}(S, T)) \right) \wedge_{Bool} (\neg_{Bool} (T ==_K \text{float})) \right) \wedge_{Bool} (\neg_{Bool} \text{isArrayType}(\text{t}(S, T))) \right) \vee_{Bool} (\text{rank}(\text{t}(S, T)) \geq_{Int} \text{rank}(\text{t}(\bullet, \text{int}))) \right) \wedge_{Bool} (T ==_K \text{double}) \vee_{Bool} (T ==_K \text{long-double})$$

DEFINE

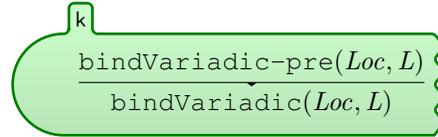
$$\frac{\text{promoteList}(\bullet)}{\bullet}$$

SYNTAX $K ::= \text{bindVariadic}(K, \text{List}\{K\text{Result}\})$
 $\quad \quad \quad | \text{bindVariadic-pre}(K, \text{List}\{K\})$

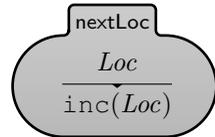
RULE BIND-VARIADIC-PRE



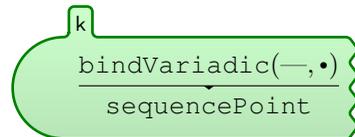
RULE BIND-VARIADIC-START



RULE BIND-VARIADIC



RULE BIND-VARIADIC-DONE

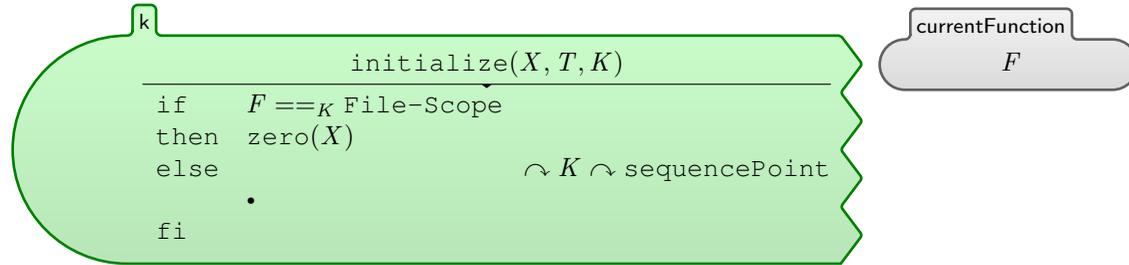


END MODULE

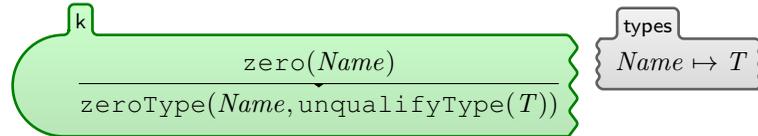
MODULE DYNAMIC-SEMANTICS-DECLARATIONS-GENERAL

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-INCLUDE

RULE

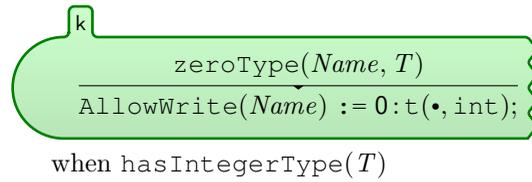


RULE

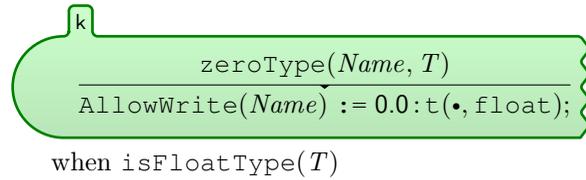


SYNTAX $K ::= \text{zeroType}(K, Type)$

RULE

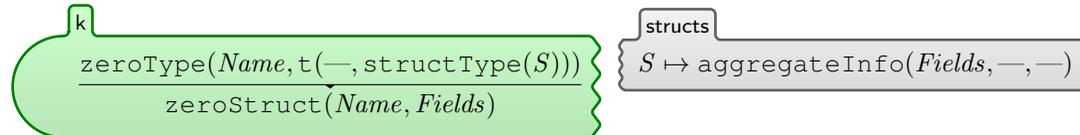


RULE



SYNTAX $K ::= \text{zeroStruct}(K, List\{KResult\})$

RULE



RULE

$$\frac{\text{zeroType}(Name, \mathfrak{t}(_, \text{unionType}(S)))}{\text{zeroType}(Name \cdot F, T)}$$

$$\text{structs } S \mapsto \text{aggregateInfo}(\text{typedDecl}(T, F), _, _, _)$$

RULE

$$\frac{\text{zeroStruct}(Name, \text{typedDecl}(T, F), L)}{\text{zeroType}(Name \cdot F, T) \curvearrowright \text{zeroStruct}(Name, L)}$$

RULE

$$\frac{\text{zeroStruct}(Name, \bullet)}{\bullet}$$

RULE

$$\frac{\text{zeroType}(Name, T)}{\text{AllowWrite}(Name) := \text{NullPointer}: T;}$$

when isPointerType(T)

RULE

$$\frac{\text{zeroType}(Name, T)}{\bullet}$$

when isFunctionType(T)

RULE

$$\frac{\text{zeroType}(Name, \mathfrak{t}(S, \text{arrayType}(T, Len)))}{\text{zeroType}(Name[Len -_{Int} 1], T) \curvearrowright \text{zeroType}(Name, \mathfrak{t}(S, \text{arrayType}(T, Len -_{Int} 1)))}$$

when $Len >_{Int} 0$

RULE

$\frac{\text{zeroType}(Name, t(-, \text{arrayType}(T, 0)))}{\cdot}$

RULE

$\frac{\text{zeroType}(Name, t(-, \text{flexibleArrayType}(T)))}{\cdot}$

END MODULE

MODULE DYNAMIC-SEMANTICS-DECLARATIONS-INITIALIZATIONS

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-INCLUDE

END MODULE

MODULE DYNAMIC-SEMANTICS-DECLARATIONS-RESOLUTION

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-INCLUDE

END MODULE

MODULE DYNAMIC-SEMANTICS-DECLARATIONS-ENUMS

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-INCLUDE

END MODULE

MODULE DYNAMIC-C-DECLARATIONS

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-INCLUDE

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-BINDING

IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-GENERAL

```
IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-INITIALIZATIONS
```

```
IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-ENUMS
```

```
IMPORTS DYNAMIC-SEMANTICS-DECLARATIONS-RESOLUTION
```

```
END MODULE
```

A.7 Memory

This section represents the low-level operations used to manipulate memory. This includes its creation, destruction, reading, and writing. It also includes the necessary bit-packing and bit-twiddling mechanisms needed to deal with bitfields.

MODULE DYNAMIC-MEMORY-INCLUDE

IMPORTS DYNAMIC-INCLUDE

SYNTAX $K ::= \text{extractBytesFromMem}(Nat, Nat)$

SYNTAX $Nat ::= \text{encodedPointer}(Int)$
| $\text{encodedFloat}(Float)$

SYNTAX $List\{K\} ::= \text{explodeToBits}(List\{K\})$ [function]
| $\text{reverseList}(List\{K\})$ [function]

SYNTAX $ListItem ::= \text{bwrite}(Nat, K)$

SYNTAX $Set ::= \text{locations}(List)$ [function]

SYNTAX $K ::= \text{read-aux}(K, K, K)$

SYNTAX $Nat ::= \text{subObject}(K, K, K)$

DEFINE
 $\frac{\text{isInt}(\text{subObject}(_, _, _))}{\text{true}}$

DEFINE
 $\frac{\text{isInt}(\text{encodedPointer}(_))}{\text{true}}$

DEFINE
 $\frac{\text{isInt}(\text{encodedFloat}(_))}{\text{true}}$

SYNTAX $Nat ::= \text{getBitOffset}(Nat)$ [function]

DEFINE
 $\frac{\text{getBitOffset}(\text{loc}(_, _, M))}{M \%_{Int} \text{ numBitsPerByte}}$

SYNTAX $Nat ::= \text{getByteOffset}(Nat)$ [function]

```

DEFINE
   $\frac{\text{getBytesOffset}(\text{loc}(\_, M, N))}{M +_{Int} (N \div_{Int} \text{numBitsPerByte})}$ 
DEFINE LOCATIONS-NONE
   $\frac{\text{locations}(\bullet)}{\bullet}$ 
DEFINE LOCATIONS-SOME
   $\frac{\text{locations}(\text{bwrite}(Loc, \_) L)}{Loc \text{ locations}(L)}$ 

```

END MODULE

MODULE DYNAMIC-SEMANTICS-READING

IMPORTS DYNAMIC-MEMORY-INCLUDE

```

SYNTAX  $K ::= \text{extractBitsFromMem}(Nat, Nat)$ 
      |  $\text{extractByteFromMem}(Nat)$ 
      |  $\text{extractBitsFromList-aux}(K, Int, Int, List\{K\})$ 

```

RULE

$$\frac{\text{extractBitsFromList}(\text{dataList}(L), N, M)}{\text{extractBitsFromList-aux}(\text{dataList}(\text{explodeToBits}(L)), N, M, \bullet)}$$

RULE

$$\frac{\text{extractBitsFromList-aux}(\text{dataList}(\text{piece}(_, 1) \text{ } L), \text{Offset}, \text{NumBits}, \bullet)}{\text{extractBitsFromList-aux}(\text{dataList}(L), \text{Offset} -_{Int} 1, \text{NumBits}, \bullet)}$$

when $\text{Offset} >_{Int} 0$

RULE

$$\frac{\text{extractBitsFromList-aux}(\text{dataList}(\text{piece}(N, 1) \text{ } L), 0, \text{NumBits}, \text{Done})}{\text{extractBitsFromList-aux}(\text{dataList}(L), 0, \text{NumBits} -_{Int} 1, \text{Done} \text{ } \text{piece}(N, 1))}$$

when $\text{NumBits} >_{Int} 0$

RULE

$$\frac{\text{extractBitsFromList-*aux*(\text{---}, 0, 0, \text{Done})}{\text{dataList}(\text{Done})}$$

CONTEXT: readActual(---, ---, value(□))

RULE

$$\frac{\text{read}(\text{Loc}, T)}{\text{read-*aux*}(\text{Loc}, T, \text{value}(\text{bitSizeofType}(T)))}$$

when $\neg_{Bool} \text{isFunctionType}(T)$

SYNTAX $K ::= \text{readActual}(K, K, K)$

These rules figure out whether the read should be structural or computational, depending on what is being read

RULE READ-THREAD-LOCAL

$$\frac{\text{read-*aux*}(\text{loc}(\text{threadId}(\text{Id}) + \text{Int } \text{---}, \text{---}, \text{---}) \text{ , } \text{---} \text{ , } \text{---})}{\text{readActual}}$$

[ndlocal]

threadId
Id

RULE READ-SHARED

$$\frac{\text{read-*aux*}(\text{loc}(\text{threadId}(0) + \text{Int } \text{---}, \text{---}, \text{---}) \text{ , } \text{---} \text{ , } \text{---})}{\text{readActual}}$$

[computational ndlocal]

RULE READ-ALLOCATED

$$\frac{\text{read-*aux*}(\text{loc}(\text{threadId}(\text{allocatedDuration}) + \text{Int } \text{---}, \text{---}, \text{---}) \text{ , } \text{---} \text{ , } \text{---})}{\text{readActual}}$$

[computational ndlocal]

RULE READ

$$\frac{k \quad \text{readActual}(Loc, T, bitSize)}{\text{concretize}(T, \text{extractBytesFromMem}(Loc, \text{bitsToBytes}(bitSize)))}$$

when $\neg_{Bool} \text{isBitfieldType}(T)$

RULE READ-BITFIELD

$$\frac{k \quad \text{readActual}(Loc, T, bitSize)}{\text{concretize}(T, \text{fillToBytes}(\text{extractBitsFromMem}(Loc, bitSize)))}$$

when $\text{isBitfieldType}(T)$

SYNTAX $K ::= \text{joinIntegerBytes}(Type, List\{K\})$

RULE

$$\frac{k \quad \text{concretize}(T, \text{dataList}(L))}{\text{joinIntegerBytes}(T, L)}$$

when $\text{hasIntegerType}(T) \wedge_{Bool} (\neg_{Bool} \text{isBitfieldType}(T))$

RULE

$$\frac{k \quad \text{concretize}(t(S, \text{bitfieldType}(T, Len)), \text{dataList}(L))}{\text{joinIntegerBytes}(t(S, \text{bitfieldType}(T, Len)), \text{reverseList}(L))}$$

RULE

$$\frac{k \quad \text{concretize}(T, \text{dataList}(\text{piece}(\text{encodedFloat}(F), Len), _))}{F : T}$$

when $\text{isFloatType}(T) \wedge_{Bool} (Len ==_{Int} \text{numBitsPerByte})$

SYNTAX $K ::= \text{joinPointerBytes}(Type, List\{K\})$
| $\text{joinPointerBytes-aux}(Type, List\{K\}, K)$

RULE

$$\frac{\text{concretize}(T, \text{dataList}(L))}{\text{joinPointerBytes}(T, L)}$$

when $\text{isPointerType}(T)$

RULE

$$\frac{\text{joinPointerBytes}(T, \text{piece}(N, Len), L)}{\text{joinPointerBytes-aux}(T, L, N)}$$

when $Len ==_{Int} \text{numBitsPerByte}$

RULE

$$\frac{\text{joinPointerBytes-aux}(T, \text{piece}(\text{subObject}(N, sNatEnd, sNatEnd), Len), L, \text{subObject}(N, 0, End))}{\text{joinPointerBytes-aux}(T, L, \text{subObject}(N, 0, End +_{Int} 1))}$$

when $(Len ==_{Int} \text{numBitsPerByte}) \wedge_{Bool} (sNatEnd ==_{Int} (End +_{Int} 1))$

RULE

$$\frac{\text{joinPointerBytes-aux}(T, \bullet, \text{subObject}(N, 0, End))}{\text{checkValidLoc}(N) \curvearrowright N : T}$$

RULE

$$\frac{\text{concretize}(t(S, \text{structType}(S)), \text{dataList}(L))}{L : t(S, \text{structType}(S))}$$

RULE

$$\frac{\text{concretize}(t(S, \text{unionType}(S)), \text{dataList}(L))}{L : t(S, \text{unionType}(S))}$$

SYNTAX $K ::= \text{joinIntegerBytes-aux}(Type, List\{K\}, K)$

RULE JOININTEGERBYTES-START

$$\frac{\text{joinIntegerBytes}(T, L)}{\text{joinIntegerBytes-aux}(T, L, \text{piece}(0, 0))}$$

RULE JOININTEGERBYTES-UNKNOWN-CHAR

$$\frac{\text{joinIntegerBytes-aux}(T, \text{piece}(\text{unknown}(Len), Len), \text{piece}(0, 0))}{\text{piece}(\text{unknown}(Len), Len) : T}$$

when isCharType(T)

RULE JOININTEGERBYTES-STEP

$$\frac{\text{joinIntegerBytes-aux}(T, L, \text{piece}(N, Len), \text{piece}(N', Len'))}{\text{joinIntegerBytes-aux}(T, L, \text{piece}(\text{piece}(N', Len') \text{ bit} :: \text{piece}(N, Len), Len +_{Int} Len'))}$$

when $N' \geq_{Int} 0$

RULE JOININTEGERBYTES-DONE

$$\frac{\text{joinIntegerBytes-aux}(T, \bullet, \text{piece}(N, Len))}{\text{interpret}(T, \text{piece}(N, Len))}$$

when $N \geq_{Int} 0$

DEFINE

$$\frac{\text{floorLoc}(\text{loc}(Base, Offset, BitOffset))}{\text{loc}(Base, Offset, 0)}$$

when $BitOffset <_{Int} \text{numBitsPerByte}$

DEFINE CEILINGLOC-NULL

$$\frac{\text{ceilingLoc}(\text{NullPointer})}{\text{NullPointer}}$$

DEFINE CEILINGLOC

$$\frac{\text{ceilingLoc}(\text{loc}(N, R, M))}{\text{loc}(N, (M \div_{Int} \text{numBitsPerByte}) +_{Int} R, 0)}$$

RULE

$$\frac{\text{extractBitsFromMem}(Loc, Size)}{\text{extractBitsFromList}(\text{extractBytesFromMem}(\text{floorLoc}(Loc), \text{bitsToBytes}(Size +_{Int} \text{getBitOffset}(Loc))), \text{getBitOffset}(Loc), Size)}$$

SYNTAX $K ::= \text{extractBytesFromMem-aux}(K, K, \text{List}\{K\})$

RULE

$$\frac{\text{extractBytesFromMem}(Loc, Size)}{\text{extractBytesFromMem-aux}(Loc, Size, \bullet)}$$

RULE

$$\frac{\text{extractBytesFromMem-aux}(Loc, Size, Aux)}{\text{extractByteFromMem}(Loc) \curvearrowright \text{extractBytesFromMem-aux}(Loc +_{Int} 1, Size -_{Int} 1, Aux)}$$

when $Size >_{Int} 0$

RULE

$$\frac{V : T \curvearrowright \text{extractBytesFromMem-aux}(_, _, \frac{Aux}{Aux, V : T})}{\bullet}$$

SYNTAX $List\{K\} ::= \text{values}(List\{K\})$ [function]

DEFINE

$$\frac{\text{values}(K : _, L)}{K, \text{values}(L)}$$

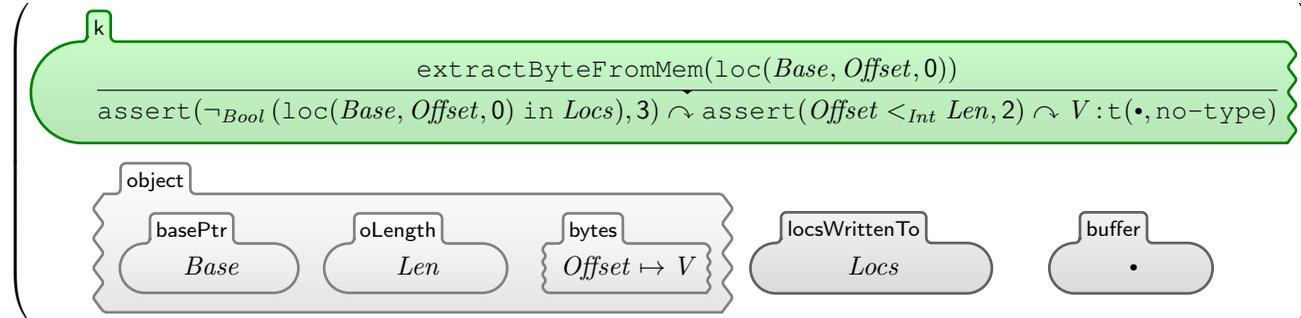
DEFINE

$$\frac{\text{values}(\bullet)}{\bullet}$$

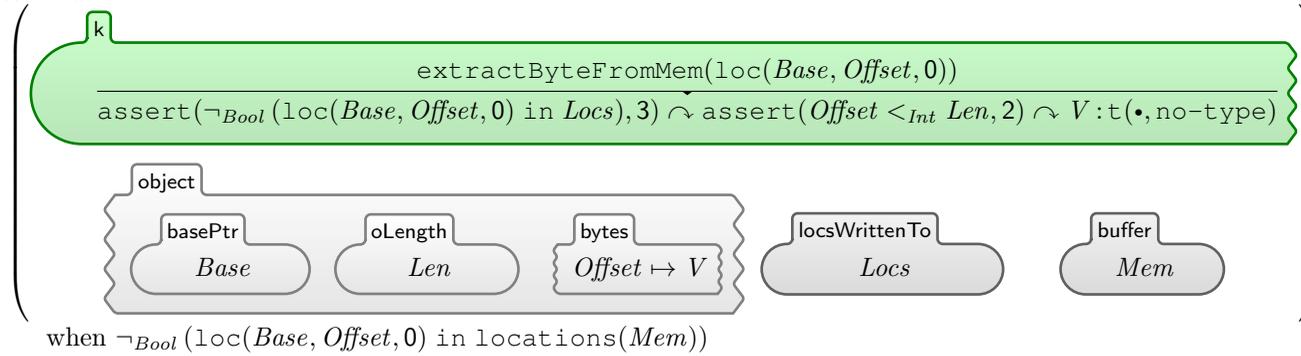
RULE

$$\frac{\text{extractBytesFromMem-aux}(_, 0, Aux)}{\text{dataList}(\text{values}(Aux))}$$

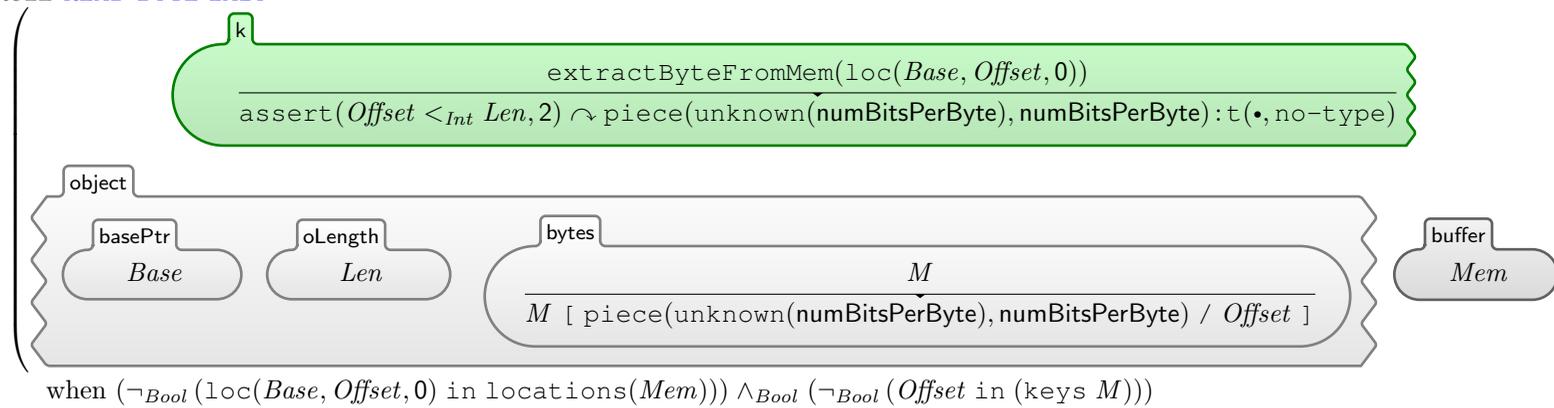
RULE READ-BYTE-FAST



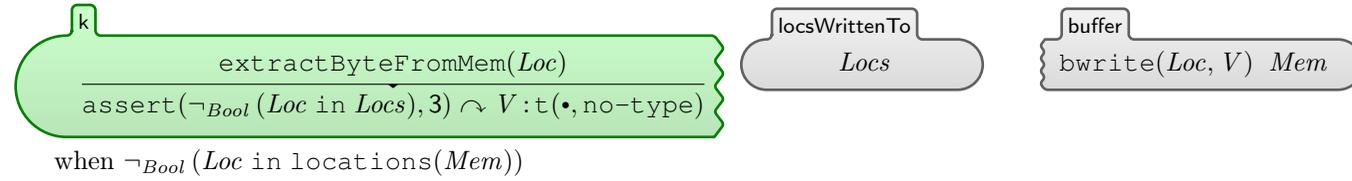
RULE READ-BYTE



RULE READ-BYTE-LAZY



RULE READ-BYTE-BUFFER



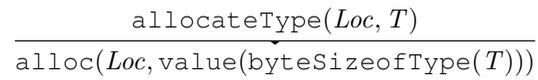
END MODULE

MODULE DYNAMIC-SEMANTICS-WRITING

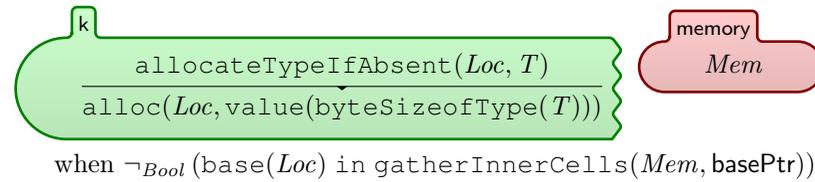
IMPORTS DYNAMIC-MEMORY-INCLUDE

CONTEXT: alloc(—, value(□))

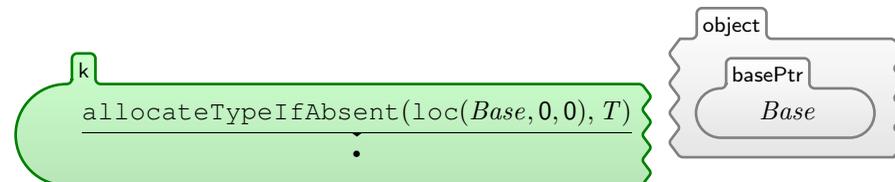
RULE



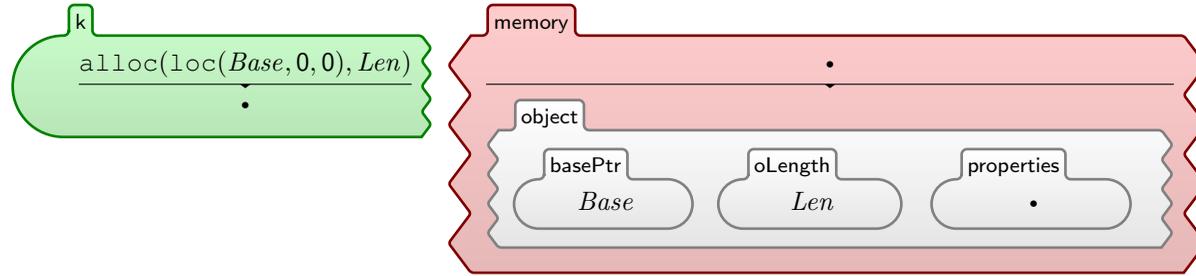
RULE ALLOCATETYPEIFABSENT-ABSENT



RULE ALLOCATETYPEIFABSENT-PRESENT

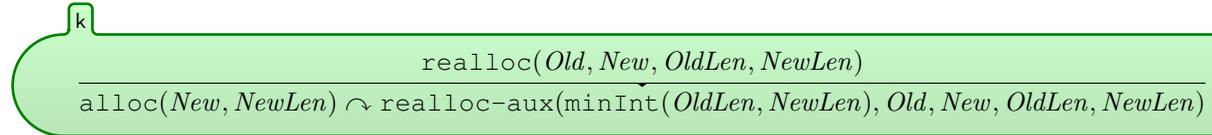


RULE ALLOC-LAZY

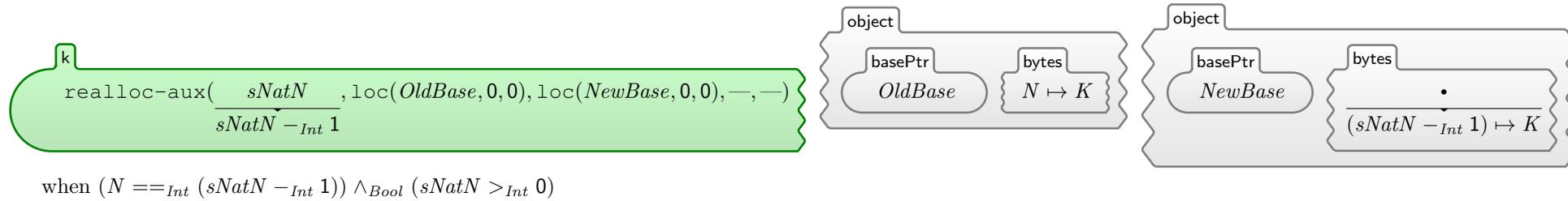


SYNTAX $K ::= \text{realloc-aux}(K, K, K, K, K)$

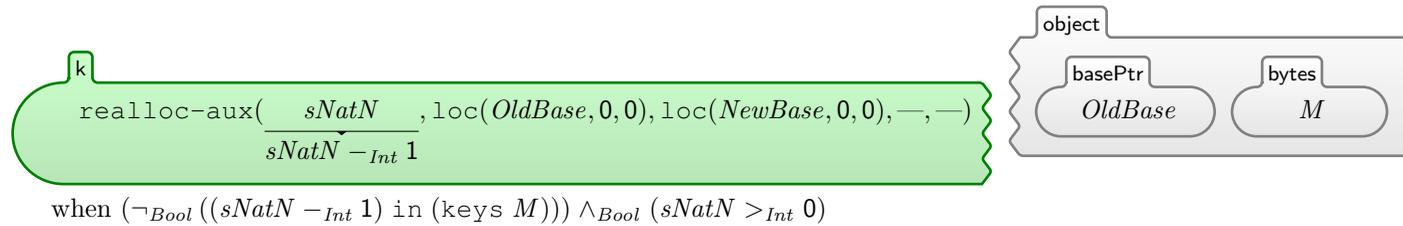
RULE REALLOC-START



RULE REALLOC-FOUND



RULE REALLOC-UNFOUND



RULE **REALLOC-0**

k

$$\frac{\text{realloc-aux}(0, \text{loc}(\text{OldBase}, 0, 0), \text{---}, \text{OldLen}, \text{---})}{\text{deleteSizedBlock}(\text{loc}(\text{OldBase}, 0, 0), \text{OldLen})}$$

SYNTAX $K ::= \text{writeBytes}(\text{Nat}, K)$ [strict(2)]
| $\text{writeBitfield}(\text{Nat}, \text{Type}, K)$ [strict(3)]

SYNTAX $\text{Bool} ::= \text{isByteLoc}(\text{Nat})$ [function]

SYNTAX $K ::= \text{splitBytes}(\text{Value})$ [function]
| $\text{calcNewBytes}(\text{Nat}, K, K)$ [function strict(3)]
| $\text{write-aux}(K, \text{Value}, K)$ [strict(2)]

RULE

$$\frac{\text{write}(\text{lv}(\text{Dest}, T'), V : T)}{\text{write-aux}(\text{Dest}, V : T, \text{value}(\text{bitSizeofType}(T)))}$$

when $\neg_{\text{Bool}} \text{isConstType}(T')$

CONTEXT: $\text{write-aux}(\text{---}, \text{---}, \text{value}(\square))$

SYNTAX $K ::= \text{write-specific}(\text{Nat}, \text{Value}, \text{Nat})$

RULE **WRITE-THREAD-LOCAL**

k

$$\frac{\text{write-aux}(\text{loc}(\text{threadId}(\text{Id}) +_{\text{Int}} N, \text{Offset}, \text{BitOffset}), L : T, \text{bitSize})}{\text{write-specific}(\text{loc}(\text{threadId}(\text{Id}) +_{\text{Int}} N, \text{Offset}, \text{BitOffset}), L : T, \text{bitSize})}$$

threadId

Id

[ndlocal]

RULE **WRITE**

k

$$\frac{\text{write-aux}(\text{loc}(\text{threadId}(0) +_{\text{Int}} N, \text{Offset}, \text{BitOffset}), L : T, \text{bitSize})}{\text{write-specific}(\text{loc}(\text{threadId}(0) +_{\text{Int}} N, \text{Offset}, \text{BitOffset}), L : T, \text{bitSize})}$$

[computational ndlocal]

RULE WRITE-ALLOCATED

$$\frac{\text{write-aux}(\text{loc}(\text{threadId}(\text{allocatedDuration}) +_{Int} N, \text{Offset}, \text{BitOffset}), L : T, \text{bitSize})}{\text{write-specific}(\text{loc}(\text{threadId}(\text{allocatedDuration}) +_{Int} N, \text{Offset}, \text{BitOffset}), L : T, \text{bitSize})}$$

[computational ndlocal]

RULE WRITE-NORMAL

$$\frac{\text{write-specific}(Loc, V : T, \text{bitSize})}{\text{writeBytes}(Loc, \text{splitBytes}(V : T))}$$

when $((\text{bitSize} \%_{Int} \text{numBitsPerByte}) ==_{Int} 0) \wedge_{Bool} \text{isByteLoc}(Loc) \wedge_{Bool} (\neg_{Bool} \text{isBitFieldType}(T)) \wedge_{Bool} (\neg_{Bool} \text{hasUnionMarker}(T))$

RULE WRITE-NORMAL-UNION-FIELD

$$\frac{\bullet \quad \text{write-specific}(Loc, \text{t}(\text{fromUnion}(S) \text{ } \text{---}, \text{---}), \text{---})}{\text{makeUnknown}(Loc, \text{t}(\bullet, \text{unionType}(S)))}$$

when $\text{isByteLoc}(Loc)$

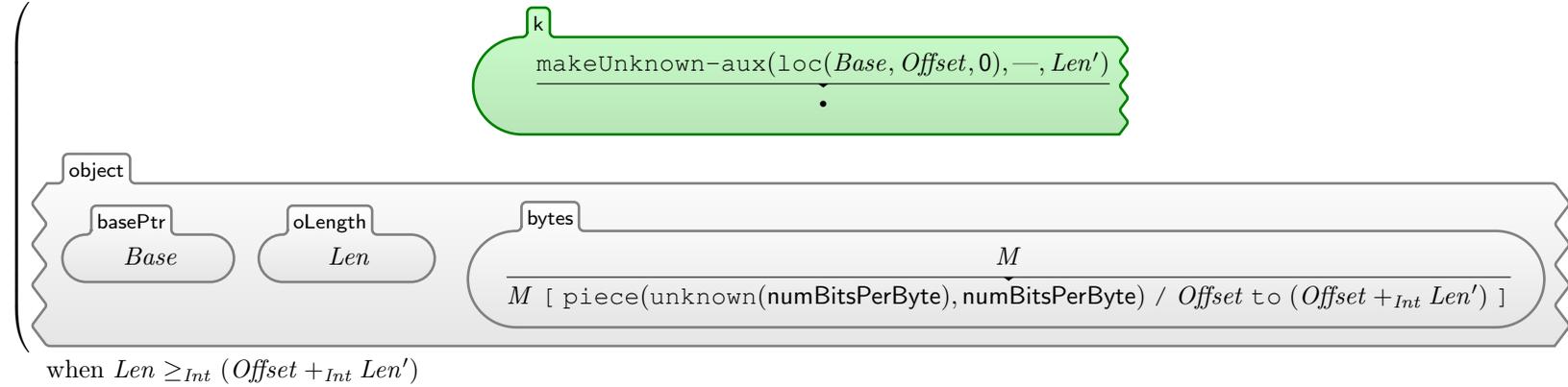
SYNTAX $K ::= \text{makeUnknown}(Nat, Type)$
 $\quad | \text{makeUnknown-aux}(Nat, Type, K)$

CONTEXT: $\text{makeUnknown-aux}(\text{---}, \text{---}, \text{value}(\square))$

RULE

$$\frac{\text{makeUnknown}(Loc, T)}{\text{makeUnknown-aux}(Loc, T, \text{value}(\text{byteSizeOfType}(T)))}$$

RULE



SYNTAX $\text{Bool} ::= \text{hasUnionMarker}(\text{Type})$ [function]

DEFINE

$\text{hasUnionMarker}(\text{t}(\text{fromUnion}(-) \text{---}))$
 \downarrow
 true

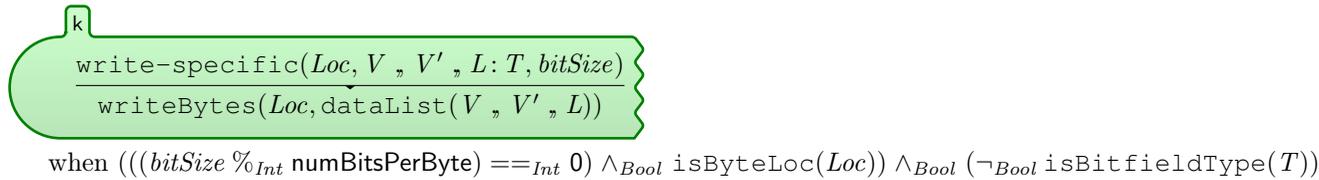
DEFINE

$\text{hasUnionMarker}(\text{t}(\bullet, -))$
 \downarrow
 false

DEFINE

$\text{hasUnionMarker}(\text{t}(S \text{ (} L(-)), K))$
 \downarrow
 $\text{hasUnionMarker}(\text{t}(S, K))$
 when $L \neq_{K\text{Label}} \text{fromUnion}$

RULE WRITE-STRUCT



SYNTAX $\text{Value} ::= \text{justBits}(\text{Int}, \text{Type})$ [function]
 | $\text{justBits-aux}(\text{Int}, \text{Type})$ [function]

DEFINE

$$\frac{\text{justBits}(I, \text{---})}{I}$$

when $I \geq_{Int} 0$

DEFINE

$$\frac{\text{justBits}(I, T)}{\text{justBits-aux}(I, T)}$$

 when $I <_{Int} 0$

DEFINE

$$\frac{\text{justBits-aux}(I, T)}{\text{justBits-aux}((I +_{Int} \text{max}(T)) +_{Int} 1, T)}$$

 when $I <_{Int} 0$

DEFINE

$$\frac{\text{justBits-aux}(N, \text{t}(\text{---}, \text{bitFieldType}(T, Len)))}{N +_{Int} (1 \ll_{Int} (Len -_{Int} 1))}$$

 when $Len >_{Int} 0$

RULE MESSAGE-BITFIELD

k

$$\frac{\text{write-specific}(Loc, I : T, bitSize)}{\text{write-specific}(Loc, \text{justBits}(I, T) : T, bitSize)}$$

when $\left(\left(\left((bitSize \%_{Int} \text{numBitsPerByte}) \neq_{Int} 0 \right) \right) \wedge_{Bool} (I <_{Int} 0) \right) \wedge_{Bool} \left(\begin{array}{l} \forall_{Bool} (\neg_{Bool} \text{isByteLoc}(Loc)) \\ \forall_{Bool} \text{isBitFieldType}(T) \end{array} \right)$

RULE WRITE-BITFIELD

k

$$\frac{\text{write-specific}(Loc, N : T, bitSize)}{\text{writeBitField}(Loc, T, \text{calcNewBytes}(\text{getBitOffset}(Loc), \text{piece}(N, bitSize), \text{extractBytesFromMem}(\text{floorLoc}(Loc), \text{bitsToBytes}(bitSize +_{Int} \text{getBitOffset}(Loc))))))}$$

when $\left(\left((bitSize \%_{Int} \text{numBitsPerByte}) \neq_{Int} 0 \right) \right) \wedge_{Bool} \left(\begin{array}{l} \forall_{Bool} (\neg_{Bool} \text{isByteLoc}(Loc)) \\ \forall_{Bool} (\text{isBitFieldType}(T) \wedge_{Bool} (\neg_{Bool} \text{hasUnionMarker}(T))) \end{array} \right)$

RULE

$$\frac{\text{writeBitfield}(Loc, \text{---}, \text{dataList}(L))}{\text{writeBytes}(\text{floorLoc}(Loc), \text{dataList}(L))}$$

SYNTAX $K ::= \text{calculateNewBytes-aux}(Int, K, K, List\{K\})$ [function]

DEFINE

$$\frac{\text{calcNewBytes}(Len, N, \text{dataList}(L))}{\text{calculateNewBytes-aux}(Len, \text{dataList}(\text{explodeToBits}(N)), \text{dataList}(\text{explodeToBits}(L)), \bullet)}$$

DEFINE

$$\frac{\text{reverseList}(\bullet)}{\bullet}$$

DEFINE

$$\frac{\text{reverseList}(K, L)}{\text{reverseList}(L), K}$$

SYNTAX $K ::= \text{joinBitsToBytes}(List\{K\})$ [function]
| $\text{joinBitsToBytes-aux}(K, K)$ [function]

DEFINE

$$\frac{\text{calculateNewBytes-aux}(N, K, \text{dataList}(\text{piece}(Bit, 1), L), Result)}{\text{calculateNewBytes-aux}(N -_{Int} 1, K, \text{dataList}(L), Result, \text{piece}(Bit, 1))}$$

when $N >_{Int} 0$

DEFINE

$$\frac{\text{calculateNewBytes-aux}(0, \text{dataList}(\text{piece}(N, 1), L), \text{dataList}(\text{piece}(\text{---}, 1), L'), Result)}{\text{calculateNewBytes-aux}(0, \text{dataList}(L), \text{dataList}(L'), Result, \text{piece}(N, 1))}$$

DEFINE

$$\frac{\text{calculateNewBytes-aux}(0, \text{dataList}(\bullet), \text{dataList}(L), Result)}{\text{joinBitsToBytes}(Result, L)}$$

DEFINE

$$\frac{\text{joinBitsToBytes}(L)}{\text{joinBitsToBytes-aux}(\text{dataList}(L), \text{dataList}(\bullet))}$$

```

DEFINE
  
$$\frac{\text{joinBitsToBytes-aux}(\text{dataList}(\text{piece}(N, Len) \text{ , } \text{piece}(M, 1) \text{ , } L), \text{dataList}(R))}{\text{joinBitsToBytes-aux}(\text{dataList}(\text{piece}(\text{piece}(N, Len) \text{ bit}::\text{piece}(M, 1), Len +_{Int} 1) \text{ , } L), \text{dataList}(R))}$$

  when  $Len <_{Int} \text{numBitsPerByte}$ 

DEFINE
  
$$\frac{\text{joinBitsToBytes-aux}(\text{dataList}(\text{piece}(N, Len) \text{ , } L), \text{dataList}(R))}{\text{joinBitsToBytes-aux}(\text{dataList}(L), \text{dataList}(R \text{ , } \text{piece}(N, Len)))}$$

  when  $Len ==_{Int} \text{numBitsPerByte}$ 

DEFINE
  
$$\frac{\text{joinBitsToBytes-aux}(\text{dataList}(\bullet), \text{dataList}(R))}{\text{dataList}(R)}$$


DEFINE
  
$$\frac{\text{explodeToBits}(K \text{ , } L)}{\text{explodeToBits}(K) \text{ , } \text{explodeToBits}(L)}$$


DEFINE
  
$$\frac{\text{explodeToBits}(\text{piece}(N, Len))}{\text{splinter}(N, Len)}$$

  when  $Len >_{Int} 0$ 

DEFINE
  
$$\frac{\text{explodeToBits}(\text{piece}(N, 0))}{\bullet}$$


DEFINE
  
$$\frac{\text{explodeToBits}(\bullet)}{\bullet}$$


SYNTAX  $List\{K\} ::= \text{splinter}(Nat, Nat) [\text{function}]$ 
      |  $\text{splinter-aux}(Nat, Nat, Nat) [\text{function}]$ 

DEFINE
  
$$\frac{\text{splinter}(N, Len)}{\text{splinter-aux}(N, Len, 0)}$$


DEFINE
  
$$\frac{\text{splinter-aux}(-, Len, Len)}{\bullet}$$


```

DEFINE

$$\frac{\text{splinter-aux}(N, Len, Pos)}{\text{splinter-aux}(N, Len, Pos +_{Int} 1) \text{ , piece}(\text{bitRange}(N, Pos, Pos), 1) \text{ when } Pos <_{Int} Len}$$

RULE

$$\frac{\text{writeBytes}(Loc, \text{dataList}(V \text{ , } L))}{\text{writeByte}(Loc, V) \curvearrow \text{writeBytes}(Loc +_{Int} 1, \text{dataList}(L))}$$

RULE WRITE-BYTE-BUFFER

$$\left(\begin{array}{c} \frac{\text{writeByte}(\text{loc}(Base, Offset, 0), V)}{\bullet} \quad \frac{\text{buffer}}{\text{bwrite}(\text{loc}(Base, Offset, 0), V)} \\ \frac{\text{object}}{\text{basePtr } Base \quad \text{oLength } Len \quad \text{properties } Attr} \quad \frac{\text{locsWrittenTo } Locs}{\text{loc}(Base, Offset, 0)} \quad \frac{\text{notWritable}}{Not\Writable} \end{array} \right)$$

when $((\neg_{Bool}(\text{loc}(Base, Offset, 0) \text{ in } Locs)) \wedge_{Bool} (Offset <_{Int} Len)) \wedge_{Bool} (\neg_{Bool}(\text{mconst} \text{ in } Attr)) \wedge_{Bool} (\neg_{Bool}(\text{loc}(Base, Offset, 0) \text{ in } (\text{keys } NotWritable)))$

RULE COMMIT-BYTE

$$\frac{\text{bwrite}(\text{loc}(Base, Offset, 0), V)}{\bullet} \quad \frac{\text{object}}{\text{basePtr } Base \quad \text{oLength } Len \quad \text{bytes } M \quad \frac{M}{M [V / Offset]}}$$

when $Offset <_{Int} Len$

RULE

$$\frac{\text{writeBytes}(\text{---}, \text{dataList}(\bullet))}{\bullet}$$

```

DEFINE SPLITBYTES-CHAR
  
$$\frac{\text{splitBytes}(N : T)}{\text{dataList}(\text{piece}(N, \text{numBitsPerByte}))}$$

  when isCharType( $T$ )

DEFINE SPLITBYTES-INT
  
$$\frac{\text{splitBytes}(I : T)}{\text{splitIntegerBytes}(I, T, \text{bitsToBytes}(\text{value}(\text{bitSizeOfType}(T))))}$$

  when hasIntegerType( $T$ )  $\wedge_{Bool} \left( \begin{array}{l} (I \geq_{Int} 0) \\ \vee_{Bool} (I \leq_{Int} 0) \end{array} \right)$ 

DEFINE SPLITBYTES-FLOAT
  
$$\frac{\text{splitBytes}(F : T)}{\text{splitFloatBytes}(F, T, \text{value}(\text{byteSizeOfType}(T)))}$$

  when isFloatType( $T$ )

DEFINE SPLITBYTES-POINTER
  
$$\frac{\text{splitBytes}(I : t(S, \text{pointerType}(T)))}{\text{splitPointerBytes}(I, t(S, \text{pointerType}(T)), \text{value}(\text{byteSizeOfType}(t(\bullet, \text{pointerType}(T)))))}$$


DEFINE SPLITBYTES-STRUCT
  
$$\frac{\text{splitBytes}(L : t(S, \text{structType}(S)))}{\text{splitStructBytes}(\text{dataList}(L), t(S, \text{structType}(S)), \text{value}(\text{byteSizeOfType}(t(S, \text{structType}(S)))))}$$


DEFINE SPLITBYTES-UNION
  
$$\frac{\text{splitBytes}(L : t(S, \text{unionType}(S)))}{\text{splitStructBytes}(\text{dataList}(L), t(S, \text{unionType}(S)), \text{value}(\text{byteSizeOfType}(t(S, \text{unionType}(S)))))}$$


SYNTAX  $K ::= \text{splitIntegerBytes}(K, K, K)$  [function]
      |  $\text{splitIntegerBytes-aux}(K, K, K, \text{List}\{K\})$  [function]

DEFINE
  
$$\frac{\text{splitIntegerBytes}(I, T, Len)}{\text{splitIntegerBytes-aux}(I, T, Len, \bullet)}$$

DEFINE
  
$$\frac{\text{splitIntegerBytes-aux}(I, T, Len, L)}{\text{splitIntegerBytes-aux}(I \gg_{Int} \text{numBitsPerByte}, T, Len -_{Int} 1, L, \text{lowestByte}(I, T))}$$

  when  $Len >_{Int} 0$ 

```

DEFINE

$$\frac{\text{splitIntegerBytes-aux}(-, -, 0, L)}{\text{dataList}(L)}$$

SYNTAX $K ::= \text{splitStructBytes}(K, K, K)$ [function]
 | $\text{splitStructBytes}(K, K, K, \text{List}\{K\})$ [function]

CONTEXT: $\text{splitStructBytes}(-, -, \text{value}(\square))$

DEFINE

$$\frac{\text{splitStructBytes}(\text{dataList}(L), T, Len)}{\text{splitStructBytes}(\text{dataList}(L), T, Len, \bullet)}$$

DEFINE

$$\frac{\text{splitStructBytes}(\text{dataList}(\text{piece}(N, \text{PieceLen}), Rest), T, Len, L)}{\text{splitStructBytes}(\text{dataList}(Rest), T, Len -_{Int} 1, L, \text{piece}(N, \text{PieceLen}))}$$

$$\text{when } (PieceLen ==_{Int} \text{numBitsPerByte}) \wedge_{Bool} (Len >_{Int} 0)$$

DEFINE

$$\frac{\text{splitStructBytes}(-, -, 0, L)}{\text{dataList}(L)}$$

SYNTAX $K ::= \text{splitPointerBytes}(K, K, K)$ [function]
 | $\text{splitPointerBytes-aux}(K, K, K, K, \text{List}\{K\})$ [function]

CONTEXT: $\text{splitPointerBytes}(-, -, \text{value}(\square))$

DEFINE

$$\frac{\text{splitPointerBytes}(I, T, Len)}{\text{splitPointerBytes-aux}(I, T, Len, 0, \bullet)}$$

DEFINE

$$\frac{\text{splitPointerBytes-aux}(I, T, Len, N, L)}{\text{splitPointerBytes-aux}(I, T, Len -_{Int} 1, N +_{Int} 1, L, \text{piece}(\text{subObject}(I, N, N), \text{numBitsPerByte}))}$$

$$\text{when } Len >_{Int} 0$$

DEFINE

$$\frac{\text{splitPointerBytes-aux}(-, -, 0, -, L)}{\text{dataList}(L)}$$

SYNTAX $K ::= \text{splitFloatBytes}(K, K, K)$ [function]
 | $\text{splitFloatBytes}(K, K, K, \text{List}\{K\})$ [function]

CONTEXT: splitFloatBytes(—, —, value(□))

DEFINE

$$\frac{\text{splitFloatBytes}(F, T, Len)}{\text{splitFloatBytes}(F, T, Len -_{Int} 1, \text{piece}(\text{encodedFloat}(F), \text{numBitsPerByte}))}$$

when $Len >_{Int} 0$

DEFINE

$$\frac{\text{splitFloatBytes}(F, T, Len, L)}{\text{splitFloatBytes}(F, T, Len -_{Int} 1, L, \text{piece}(\text{unknown}(\text{numBitsPerByte}), \text{numBitsPerByte}))}$$

when $Len >_{Int} 0$

DEFINE

$$\frac{\text{splitFloatBytes}(-, T, 0, L)}{\text{dataList}(L)}$$

SYNTAX $K ::= \text{lowestByte}(Int, Type)$ [function]

DEFINE

$$\frac{\text{lowestByte}(I, T)}{\text{piece}(I \ \&_{Int} \ \text{byteMaskSet}, \text{numBitsPerByte})}$$

when $\text{hasIntegerType}(T)$

SYNTAX $Nat ::= \text{byteMaskSet}$ [function]

DEFINE

$$\frac{\text{byteMaskSet}}{(2 \ ^{Int} \ \text{numBitsPerByte}) -_{Int} 1}$$

DEFINE

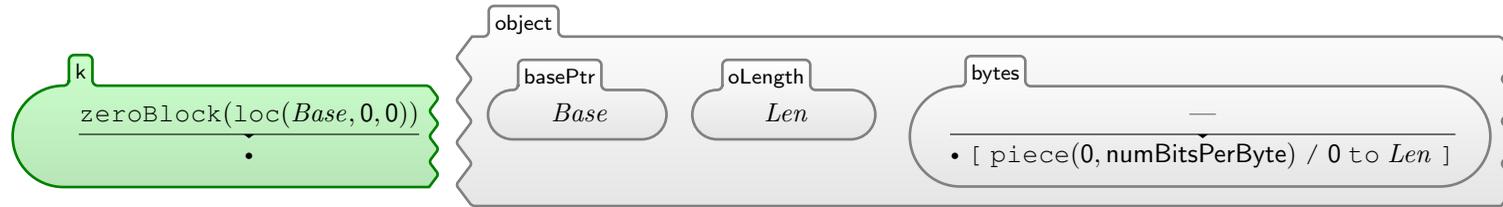
$$\frac{\text{isByteLoc}(Loc)}{\text{getBitOffset}(Loc) ==_{Int} 0}$$

END MODULE

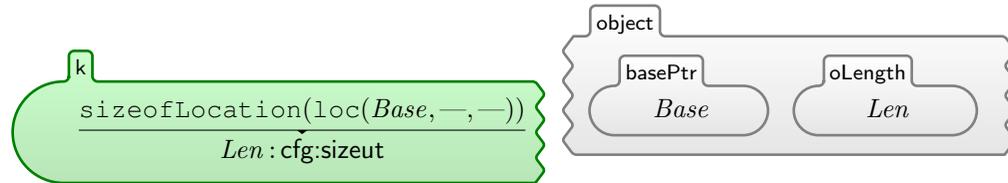
MODULE DYNAMIC-C-MEMORY-MISC

IMPORTS DYNAMIC-MEMORY-INCLUDE

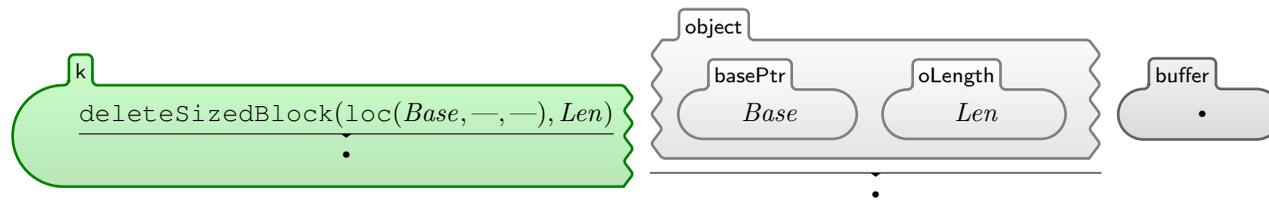
RULE



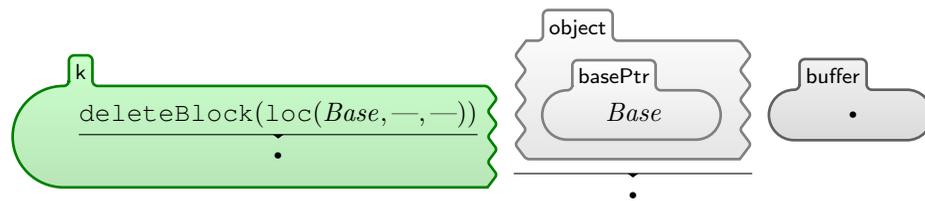
RULE



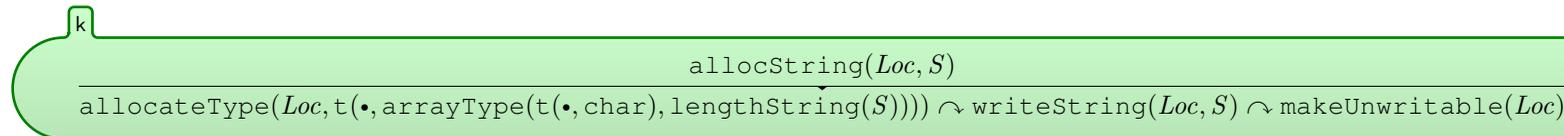
RULE DELETE-SIZED-BLOCK



RULE DELETE-BLOCK



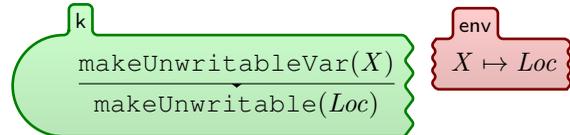
RULE ALLOC-STRING



RULE ALLOC-WSTRING



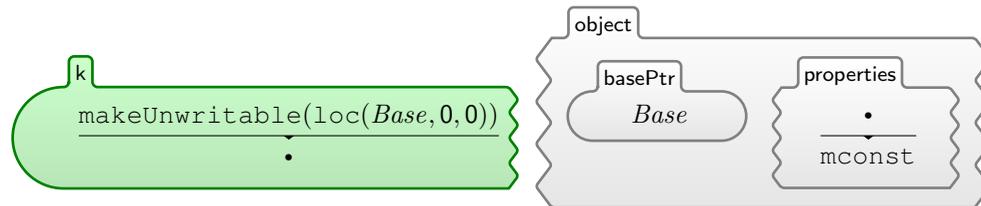
RULE MAKEUNWRITABLE-VAR



RULE MAKEUNWRITABLE-SUBOBJECT

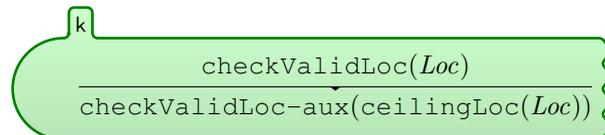


RULE MAKEUNWRITABLE



SYNTAX $K ::= \text{checkValidLoc-aux}(K)$

RULE

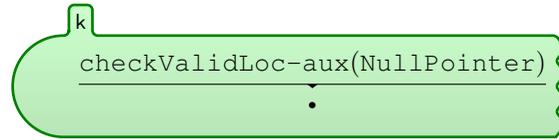


RULE

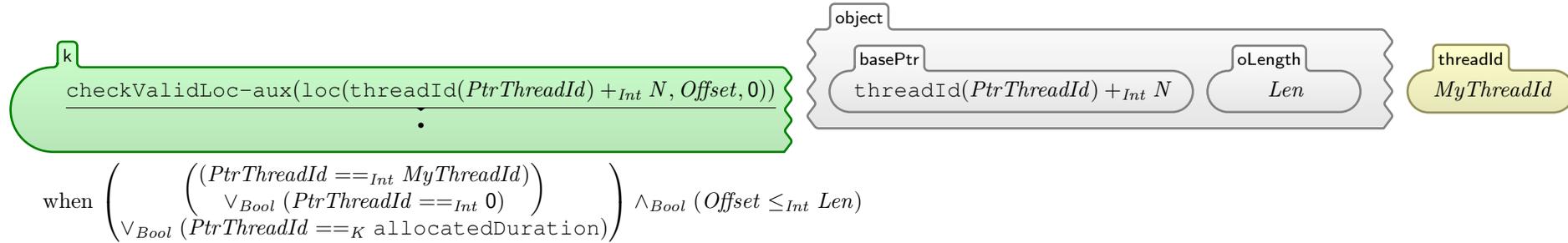


when $Loc \neq_K \text{NullPointer}$

RULE CHECK-VALID-LOC-NULL



RULE CHECK-VALID-LOC



END MODULE

MODULE DYNAMIC-C-MEMORY

IMPORTS DYNAMIC-MEMORY-INCLUDE

IMPORTS DYNAMIC-C-MEMORY-MISC

IMPORTS DYNAMIC-SEMANTICS-WRITING

IMPORTS DYNAMIC-SEMANTICS-READING

END MODULE

A.8 Standard Library

This section represents the semantics of the C standard library. It starts with some basic operators to handle reading the arguments coming in from the outside and proceeds with definitions of the actual functions.

MODULE DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

IMPORTS DYNAMIC-INCLUDE

SYNTAX $C ::= \text{vararg}(K)$ [hybrid strict]
| $\text{nextvarg}(\text{Nat}, K)$ [strict(2)]
| $\text{vpair}(K, K)$ [hybrid strict]

SYNTAX $K ::= \text{prepareBuiltin}(\text{Id}, \text{List}\{K\})$
| $\text{incSymbolic}(K)$ [function]
| $\text{printString}(K)$

END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-HELPERS

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

RULE

$$\frac{\text{reval}(\text{vararg}(K))}{\text{vararg}(K)}$$

SYNTAX $C ::= \text{nextvarg-aux}(K, \text{Type}, K, K)$

CONTEXT: $\text{nextvarg-aux}(_, _, \text{value}(\square), _)$

CONTEXT: $\text{nextvarg-aux}(_, _, _, \text{value}(\square))$

RULE NEXTVARG-START

$$\frac{\text{nextvarg}(\text{Loc}, T)}{\text{nextvarg-aux}(\text{Loc}, T, \text{value}(\text{byteSizeOfType}(T)), \text{value}(\text{sizeofLocation}(\text{Loc})))}$$

RULE NEXTVARG

$$\frac{\text{nextvarg-aux}(\text{Loc}, T, \text{Len}, \text{Len})}{\text{vpair}(\text{read}(\text{Loc}, T), \text{vararg}(\text{inc}(\text{Loc}) : \text{t}(\bullet, \text{pointerType}(\text{t}(\bullet, \text{void}))))))}$$

CONTEXT: $\text{prepareBuiltin}(-, -, \frac{\square}{\text{reval}(\square)}, -)$

SYNTAX $List\{K\} ::= \text{idsFromDeclList}(List\{K\})$ [function]

DEFINE **IDSFROMDECLLIST-ONE**

$$\frac{\text{idsFromDeclList}(L, \text{typedDecl}(t(-, T), X))}{\text{idsFromDeclList}(L), X}$$

when $T \neq_K \text{void}$

DEFINE **IDSFROMDECLLIST-VOID**

$$\frac{\text{idsFromDeclList}(L, \text{typedDecl}(t(-, \text{void}), X))}{\text{idsFromDeclList}(L)}$$

DEFINE **IDSFROMDECLLIST-VARARG**

$$\frac{\text{idsFromDeclList}(L, \text{typedDecl}(T, X), t(-, \text{variadic}))}{\text{idsFromDeclList}(L, \text{typedDecl}(T, X), \text{vararg}(\text{incSymbolic}(\text{cast}(t(\bullet, \text{pointerType}(t(\bullet, \text{unsigned-char}))), \& X)))}$$

DEFINE **IDSFROMDECLLIST-DONE**

$$\frac{\text{idsFromDeclList}(\bullet)}{\bullet}$$

CONTEXT: $\text{incSymbolic}(\frac{\square}{\text{reval}(\square)})$

DEFINE **INCSYMBOLIC**

$$\frac{\text{incSymbolic}(Loc : T)}{\text{inc}(Loc) : T}$$

RULE **PREPAREBUILTIN**

$$\frac{k}{\text{handleBuiltin}(F, t(-, \text{functionType}(\text{Return}, L))}$$

$$\text{Return}(\text{prepareBuiltin}(F, \text{idsFromDeclList}(L)))$$

RULE

$$\frac{k}{\text{printString}(S)}$$

$$\text{writeToFD}(1, \text{asciiCharString}(\text{firstChar}(S))) \curvearrowright \text{printString}(\text{butFirstChar}(S))$$

when $\text{lengthString}(S) >_{Int} 0$

RULE

k
 $\frac{\text{printString}("")}{\text{writeToFD}(1, 10)}$

END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-MATH

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

RULE **SQRT**

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"sqrt"}), F:t(-, \text{double}))}{\text{sqrtFloat}(F):t(\bullet, \text{double})}$

RULE **LOG**

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"log"}), F:t(-, \text{double}))}{\text{logFloat}(F):t(\bullet, \text{double})}$

RULE **EXP**

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"exp"}), F:t(-, \text{double}))}{\text{expFloat}(F):t(\bullet, \text{double})}$

RULE **ATAN**

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"atan"}), F:t(-, \text{double}))}{\text{atanFloat}(F):t(\bullet, \text{double})}$

RULE ASIN

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"asin"}), F : t(-, \text{double}))}{\text{asinFloat}(F) : t(\bullet, \text{double})}$$

RULE ATAN2

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"atan2"}), F : t(-, \text{double}), F' : t(-, \text{double}))}{\text{atan2Float}(F, F') : t(\bullet, \text{double})}$$

RULE TAN

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"tan"}), F : t(-, \text{double}))}{\text{tanFloat}(F) : t(\bullet, \text{double})}$$

RULE FLOOR

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"floor"}), F : t(-, \text{double}))}{\text{floorFloat}(F) : t(\bullet, \text{double})}$$

RULE COS

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"cos"}), F : t(-, \text{double}))}{\text{cosFloat}(F) : t(\bullet, \text{double})}$$

RULE FMOD

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"fmod"}), F : t(-, \text{double}), F' : t(-, \text{double}))}{F \%_{\text{Float}} F' : t(\bullet, \text{double})}$$

RULE SIN

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"sin"}), F : t(-, \text{double}))}{\text{sinFloat}(F) : t(\bullet, \text{double})}$$

END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-SETJMP

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

(n1570) §7.13 ¶1–3 The header `<setjmp.h>` defines the macro `setjmp`, and declares one function and one type, for bypassing the normal function call and return discipline.

The type declared is `jmp_buf` which is an array type suitable for holding the information needed to restore a calling environment. The environment of a call to the `setjmp` macro consists of information sufficient for a call to the `longjmp` function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point status flags, of open files, or of any other component of the abstract machine.

It is unspecified whether `setjmp` is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name `setjmp`, the behavior is undefined.

SYNTAX $K ::= \text{Bag}(\text{Bag})$
 | `ignoreLocals`

RULE **IGNORELOCALS**



(n1570) §7.13.1.1 ¶1–5 Synopsis `#include <setjmp.h>`
`int setjmp(jmp_buf env);`

Description The `setjmp` macro saves its calling environment in its `jmp_buf` argument for later use by the `longjmp` function.

Returns If the return is from a direct invocation, the `setjmp` macro returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` macro returns a nonzero value.

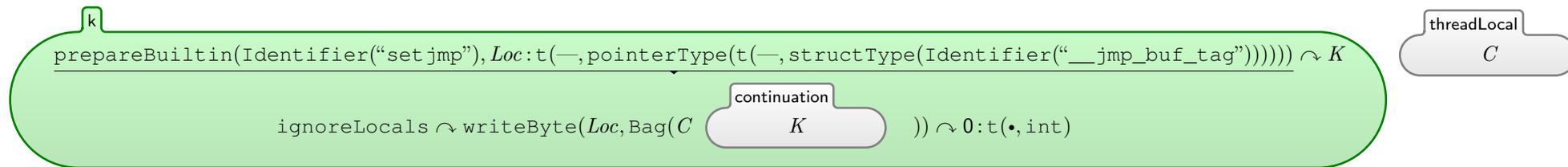
Environmental Limits An invocation of the `setjmp` macro shall appear only in one of the following contexts:

- the entire controlling expression of a selection or iteration statement;
- one operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement;
- the operand of a unary `!` operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or
- the entire expression of an expression statement (possibly cast to `void`).

If the invocation appears in any other context, the behavior is undefined.

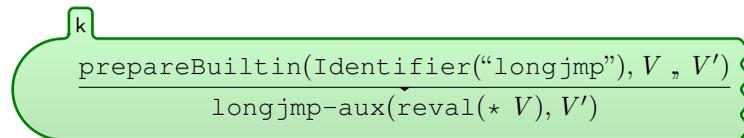
315

RULE SETJMP

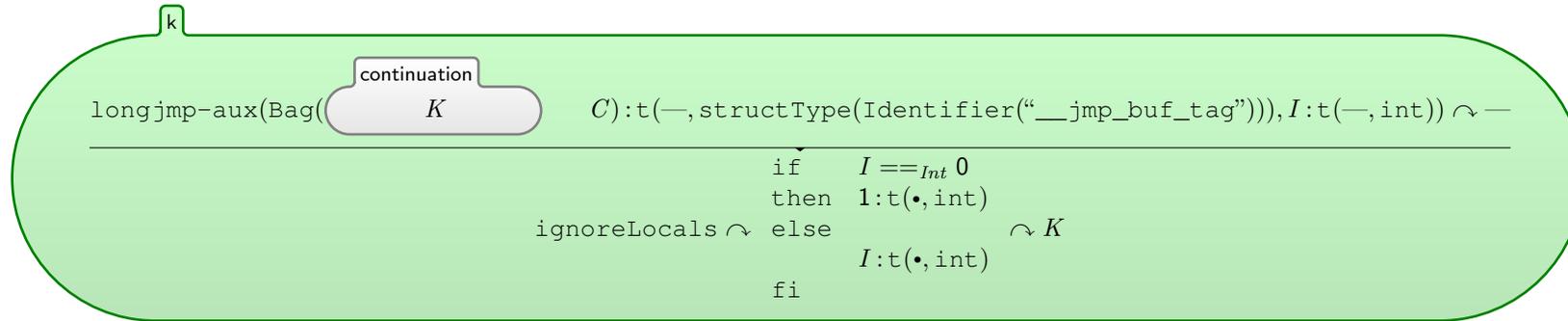


SYNTAX $K ::= \text{longjmp-aux}(K, K)$ [strict]

RULE LONGJMP-PREPARE



RULE LONGJMP



END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-STDARG

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

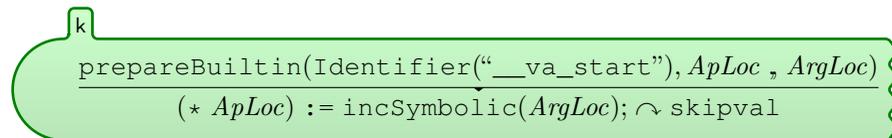
(n1570) §7.16 ¶1–3 The header <stdarg.h> declares a type and defines four macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

A function may be called with a variable number of arguments of varying types. As described in 6.9.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

The type declared is *va_list* which is a complete object type suitable for holding information needed by the macros *va_start*, *va_arg*, *va_end*, and *va_copy*. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as *ap* in this subclause) having type *va_list*. The object *ap* may be passed as an argument to another function; if that function invokes the *va_arg* macro with parameter *ap*, the value of *ap* in the calling function is indeterminate and shall be passed to the *va_end* macro prior to any further reference to *ap*.

(n1570) §7.16.1 ¶1 The *va_start* and *va_arg* macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether *va_copy* and *va_end* are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the *va_start* and *va_copy* macros shall be matched by a corresponding invocation of the *va_end* macro in the same function.

RULE VA-START



SYNTAX $K ::= \text{va-inc-aux}(K, K, K)$

CONTEXT: $\text{va-inc-aux}(-, -, \frac{\square}{\text{reval}(\square)})$

RULE

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"__va_inc"}), \text{ApLoc}, \text{Size})}{(* \text{ApLoc}) := \text{incSymbolic}(\text{ApLoc}); \curvearrowright \text{ApLoc}}$

RULE VA-INC-START

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"__va_inc"}), \text{ApLoc}, \text{Size})}{\text{va-inc-aux}(\text{ApLoc}, \text{Size}, * \text{ApLoc})}$

RULE VA-INC

k
 $\frac{\text{va-inc-aux}(\text{ApLoc}, \text{Size}, \text{Ap})}{(* \text{ApLoc}) := \text{incSymbolic}(\text{Ap}); \curvearrowright \text{Ap}}$

RULE VA-COPY

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"__va_copy"}), \text{ApLoc}, \text{Other})}{(* \text{ApLoc}) := \text{Other}; \curvearrowright \text{skipval}}$

RULE VA-END

k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"__va_end"}), \text{ApLoc})}{\text{skipval}}$

END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-STDDEF

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

SYNTAX $K ::= \text{offsetOf}(K, K)$ [strict(1)]

RULE

k
$$\frac{\text{OffsetOf}(T, K, F)}{\text{offsetOf}(\text{DeclType}(T, K), F)}$$

RULE

k
$$\frac{\text{offsetOf}(t(-, \text{structType}(S)), F)}{\text{bitsToBytes}(Offset) : \text{cfg:sizeof}}$$

 $S \mapsto \text{aggregateInfo}(-, -, - (F \mapsto Offset))$
 structs

RULE

k
$$\frac{\text{offsetOf}(t(-, \text{unionType}(-)), -)}{0 : \text{cfg:sizeof}}$$

END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-STDIO

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

RULE **PUTCHAR**

k
$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"putchar"}), N : -)}{\text{writeToFD}(1, N) \curvearrowright N : t(\bullet, \text{int})}$$

RULE **FSLPUTC**

k
$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"__fslputc"}), N : -, H : -)}{\text{writeToFD}(H, N) \curvearrowright N : t(\bullet, \text{int})}$$

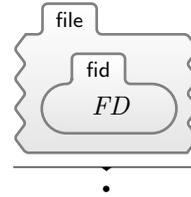
RULE **GETCHAR**

$$\frac{k \quad \text{prepareBuiltin}(\text{Identifier}(\text{"getchar"}), \bullet)}{\text{flush}(1) \curvearrowright \text{readFromFD}(0)}$$

RULE **FSLFGETC**

$$\frac{k \quad \text{prepareBuiltin}(\text{Identifier}(\text{"__fslFGetC"}), \text{FD} : - , \text{Offset} : -)}{\text{readFromFD}(\text{FD})}$$

RULE **FSLCLOSEFILE**

$$\frac{k \quad \text{prepareBuiltin}(\text{Identifier}(\text{"__fslCloseFile"}), \text{FD} : \text{t}(-, \text{int}))}{0 : \text{t}(\bullet, \text{int})}$$


RULE **FSLOPENFILE-PRE**

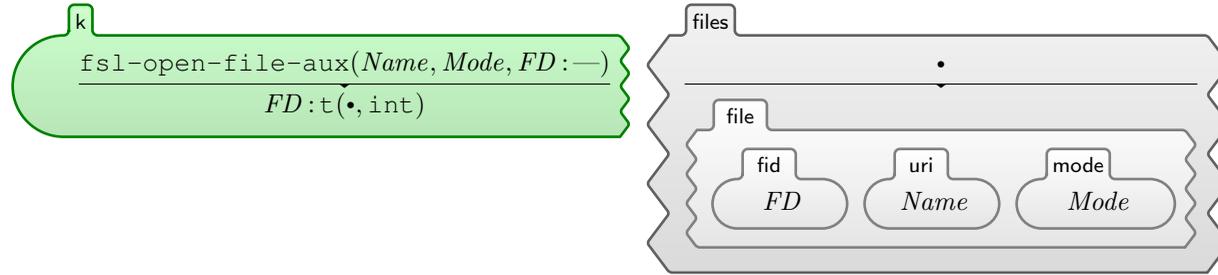
$$\frac{k \quad \text{prepareBuiltin}(\text{Identifier}(\text{"__fslOpenFile"}), \text{Filename} , \text{Mode})}{\text{fsl-open-file}(\text{getString}(\text{Filename}), \text{getString}(\text{Mode}))}$$

SYNTAX $K ::= \text{fsl-open-file-aux}(\text{String}, \text{String}, K)$ **[strict(3)]**

RULE **FSLOPENFILE-AUX**

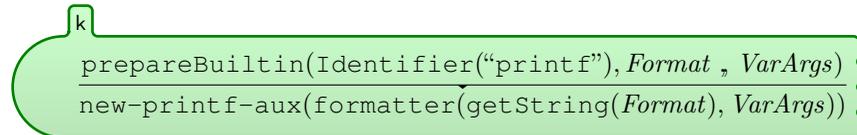
$$\frac{k \quad \text{fsl-open-file}(\text{str}(\text{Filename}), \text{str}(\text{Mode}))}{\text{fsl-open-file-aux}(\text{"file:"} + \text{String } \text{Filename}, \text{Mode}, \# \text{open}(((\text{"file:"} + \text{String } \text{Filename}) + \text{String } \text{"\#"} + \text{String } \text{Mode}))}$$

RULE **FSLOPENFILE**

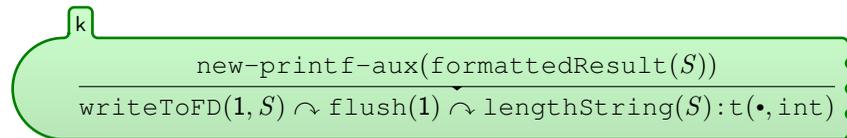


SYNTAX $K ::= \text{fsl-open-file}(K, K)$ [strict]

RULE **PRINTF**

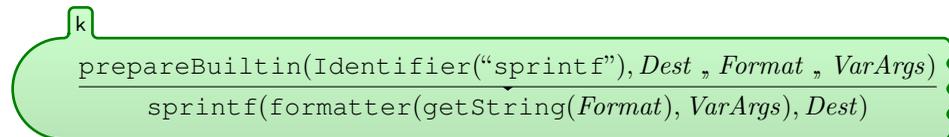


RULE **PRINTF-DONE**

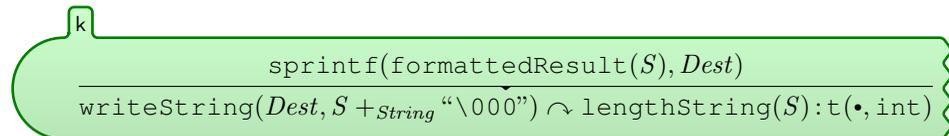


SYNTAX $K ::= \text{sprintf}(K, K)$ [strict(1)]

RULE **SPRINTF**



RULE **SPRINTF-DONE**



SYNTAX $K ::= \text{snprintf}(K, K, \text{Nat})$ [strict(1)]

RULE **SNPRINTF**

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"sprintf"}), \text{Dest}, \text{Len} : -, \text{Format}, \text{VarArgs})}{\text{sprintf}(\text{formatter}(\text{getString}(\text{Format}), \text{VarArgs}), \text{Dest}, \text{Len})}$$

RULE **SNPRINTF-DONE-NZ**

$$\frac{\text{sprintf}(\text{formattedResult}(S), \text{Dest}, \text{Len})}{\text{writeString}(\text{Dest}, \text{substrString}(S, 0, \text{Len} -_{\text{Int}} 1) +_{\text{String}} \text{"\000"}) \curvearrowright \text{lengthString}(S) : \text{t}(\bullet, \text{int})}$$

when $\text{Len} >_{\text{Int}} 0$

RULE **SNPRINTF-DONE-0**

$$\frac{\text{sprintf}(\text{formattedResult}(S), -, 0)}{\text{lengthString}(S) : \text{t}(\bullet, \text{int})}$$

SYNTAX $K ::= \text{new-printf-aux}(K) [\text{strict}]$
 $\quad | \text{formatter}(K, K) [\text{strict}(1)]$
 $\quad | \text{formatter-aux}(K) [\text{strict}]$
 $\quad | \text{formatter-next}(K)$
 $\quad | \text{formatter-arg}(K) [\text{strict}(1)]$

RULE

$$\frac{\text{formatter-next}(\text{vararg}(\text{Loc} : -))}{\text{formatter-arg}(\text{nextvarg}(\text{Loc}, \text{getFormatType}))}$$

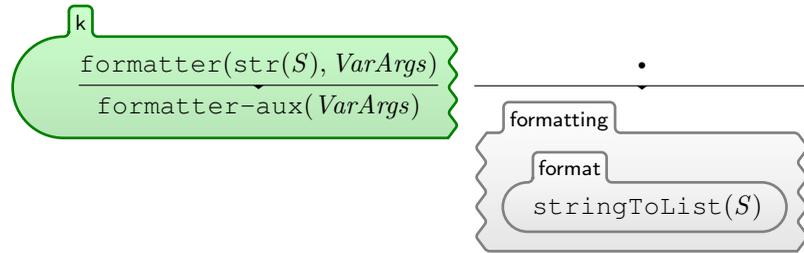
RULE

$$\frac{\text{formatter-arg}(\text{vpair}(K : -, V'))}{\text{formatter-aux}(V')}$$

$$\frac{-}{K}$$

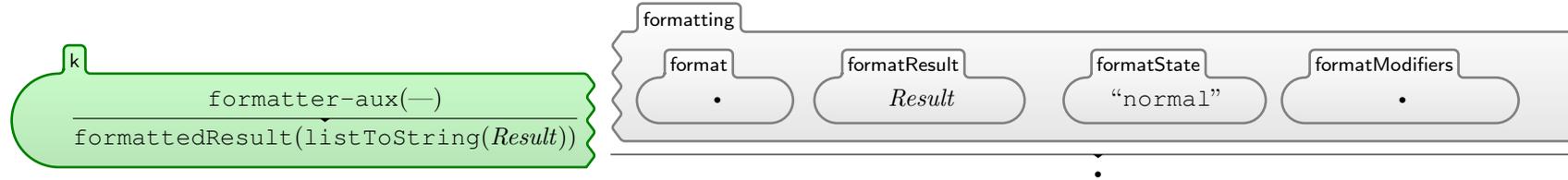
formatArg

RULE **FORMAT-START**

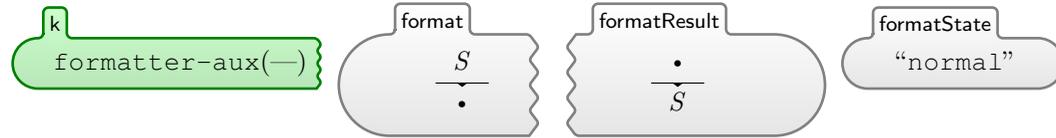


SYNTAX $\text{Value} ::= \text{formattedResult}(K)$

RULE **FORMAT-DONE**



RULE **FORMAT-NORMAL**

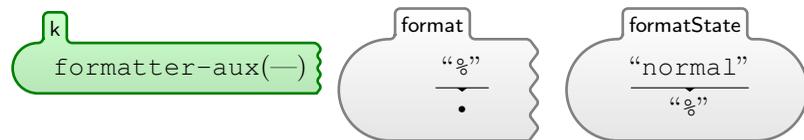


when $S \neq_{\text{String}} \text{"\%"}'$

RULE **FORMAT-RESET**



RULE **FORMAT-%**



(n1570) §7.21.6.1 ¶4 Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a nonnegative decimal integer.)
- An optional precision that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of bytes to be written for `s` conversions. The precision takes the form of a period `.` followed either by an asterisk `*` (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

(n1570) §7.21.6.1 ¶5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

(n1570) §7.21.6.1 ¶6 The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)
- + The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and + flags both appear, the *space* flag is ignored.

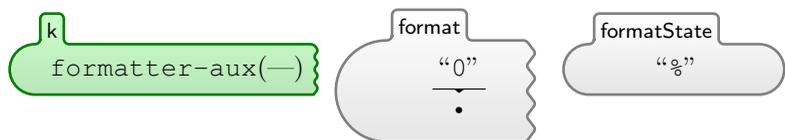
The result is converted to an “alternative form”. For o conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x (or X) conversion, a nonzero result has 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For g and G conversions, trailing zeros are *not* removed from the result. For other conversions, the behavior is undefined.

0 For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag is ignored.

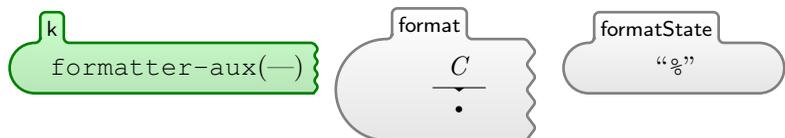
For other conversions, the behavior is undefined.

324

RULE **FORMAT-%0**



RULE **FORMAT-WIDTH**



when (charToAscii(C) >_{Int} asciiCharString(“0”)) ∧_{Bool} (charToAscii(C) ≤_{Int} asciiCharString(“9”))

(n1570) §7.21.6.1 ¶7 The length modifiers and their meanings are:

- hh** Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following `n` conversion specifier applies to a pointer to a **signed char** argument.
- h** Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short int** or **unsigned short int** before printing); or that a following `n` conversion specifier applies to a pointer to a **short int** argument.
- l (ell)** Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long int** or **unsigned long int** argument; that a following `n` conversion specifier applies to a pointer to a **long int** argument; that a following `c` conversion specifier applies to a `wint_t` argument; that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.
- ll (ell-ell)** Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long long int** or **unsigned long long int** argument; or that a following `n` conversion specifier applies to a pointer to a **long long int** argument.
- j** Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following `n` conversion specifier applies to a pointer to an `intmax_t` argument.
- z** Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to `size_t` argument.
- t** Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument.
- L** Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

SYNTAX $K ::= \text{getFormatType}$
 $\quad \quad \quad | \text{getFormatType-aux}(K, K)$

RULE



RULE

$$\frac{\text{getFormatType-aux}(\text{"\%a"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%A"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%e"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%E"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%f"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%F"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%g"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%G"}, -)}{t(\bullet, \text{double})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%c"}, \text{""})}{t(\bullet, \text{int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%s"}, \text{""})}{t(\bullet, \text{pointerType}(t(\bullet, \text{unsigned-char})))}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%p"}, \text{""})}{t(\bullet, \text{pointerType}(t(\bullet, \text{void})))}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%d"}, \text{""})}{t(\bullet, \text{int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%o"}, \text{""})}{t(\bullet, \text{int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%u"}, \text{""})}{t(\bullet, \text{unsigned-int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%x"}, \text{""})}{t(\bullet, \text{unsigned-int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%X"}, \text{""})}{t(\bullet, \text{unsigned-int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%n"}, \text{""})}{t(\bullet, \text{pointerType}(t(\bullet, \text{int})))}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%d"}, \text{"1"})}{t(\bullet, \text{long-int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%o"}, \text{"1"})}{t(\bullet, \text{long-int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%u"}, \text{"1"})}{t(\bullet, \text{unsigned-long-int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%x"}, \text{"1"})}{t(\bullet, \text{unsigned-long-int})}$$

RULE

$$\frac{\text{getFormatType-aux}(\text{"\%X"}, \text{"1"})}{t(\bullet, \text{unsigned-long-int})}$$

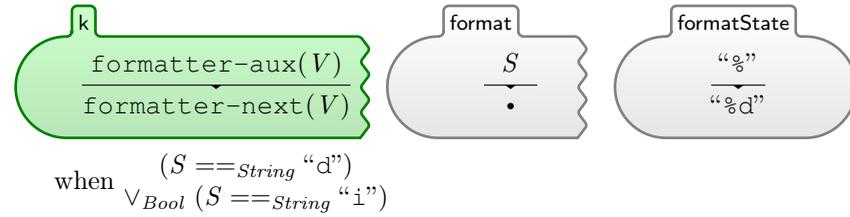
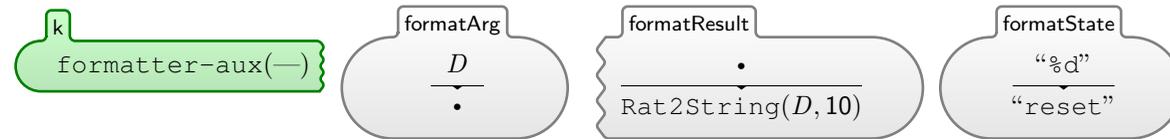
```

RULE
  getFormatType-aux("%n", "l")
  -----
  t(•, pointerType(t(•, long-int)))
RULE
  getFormatType-aux("%d", "ll")
  -----
  t(•, long-long-int)
RULE
  getFormatType-aux("%o", "ll")
  -----
  t(•, long-long-int)
RULE
  getFormatType-aux("%u", "ll")
  -----
  t(•, unsigned-long-long-int)
RULE
  getFormatType-aux("%x", "ll")
  -----
  t(•, unsigned-long-long-int)
RULE
  getFormatType-aux("%X", "ll")
  -----
  t(•, unsigned-long-long-int)
RULE
  getFormatType-aux("%n", "ll")
  -----
  t(•, pointerType(t(•, long-long-int)))

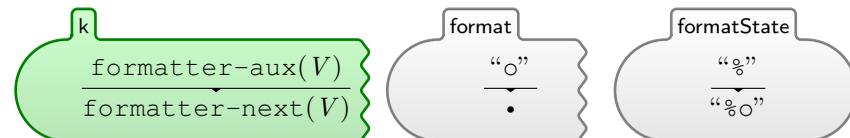
```

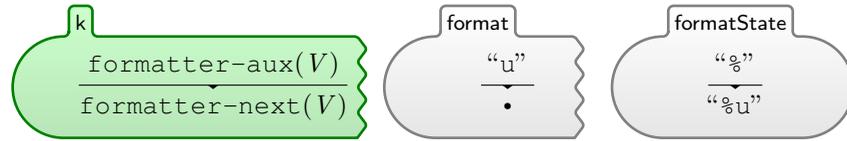
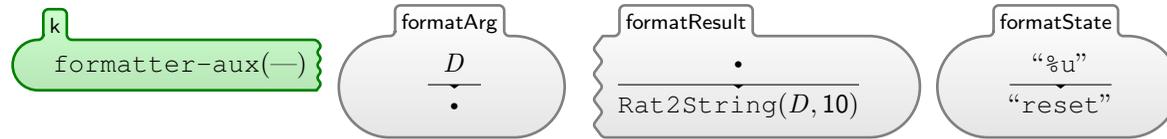
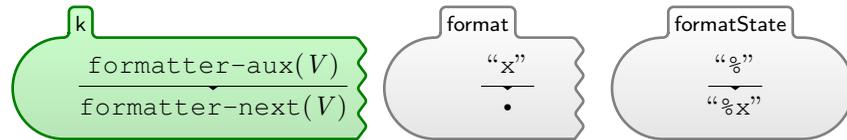
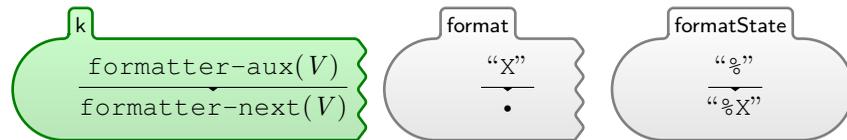
(n1570) §7.21.6.1 ¶8

d,i The **int** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

RULE **FORMAT-%D-START**RULE **FORMAT-%D****(n1570) §7.21.6.1 ¶8**

o,u,x,X The **unsigned int** argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style `ddd`; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

RULE **FORMAT-%O-START**RULE **FORMAT-%O**

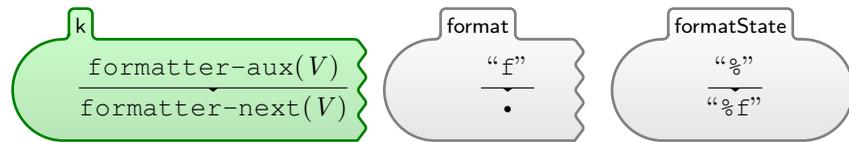
RULE **FORMAT-%U-START**RULE **FORMAT-%U**RULE **FORMAT-%X-START**RULE **FORMAT-%X**RULE **FORMAT-%X-START**RULE **FORMAT-%X**

(n1570) §7.21.6.1 ¶8

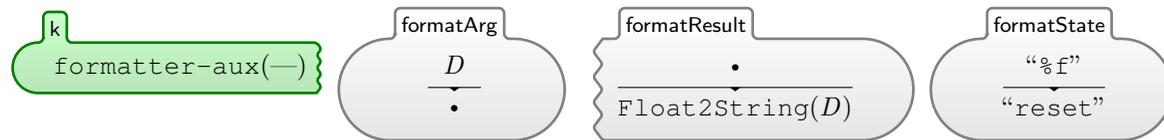
f,F A **double** argument representing a floating-point number is converted to decimal notation in the style $[-]ddd.ddd$, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A **double** argument representing an infinity is converted in one of the styles $[-]inf$ or $[-]infinity$ —which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles $[-]nan$ or $[-]nan(n-char-sequence)$ —which style, and the meaning of any $n-char-sequence$, is implementation-defined. The **F** conversion specifier produces INF, INFINITY, or NAN instead of inf , $infinity$, or nan , respectively.

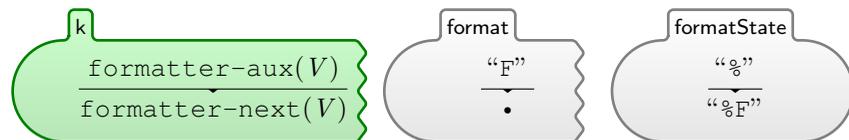
RULE **FORMAT-%F-START**



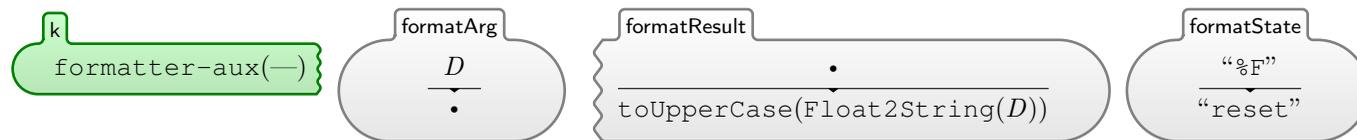
RULE **FORMAT-%F**



RULE **FORMAT-%F-START**



RULE **FORMAT-%F**



(n1570) §7.21.6.1 ¶8

e,E A **double** argument representing a floating-point number is converted in the style $[-]d.ddde\pm dd$, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The E conversion specifier produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

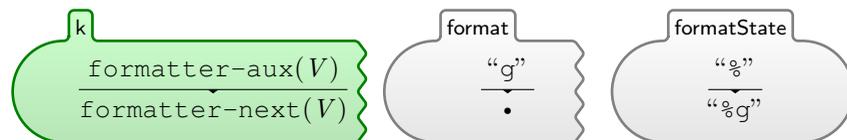
(n1570) §7.21.6.1 ¶8

g,G A **double** argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style **E** would have an exponent of X :

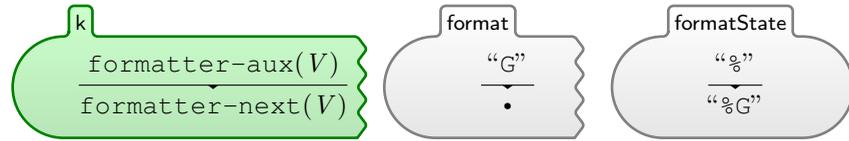
- if $P > X - 4$, the conversion is with style **f** (or **F**) and precision $P - (X + 1)$.
- otherwise, the conversion is with style **e** (or **E**) and precision $P - 1$.

Finally, unless the # flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.

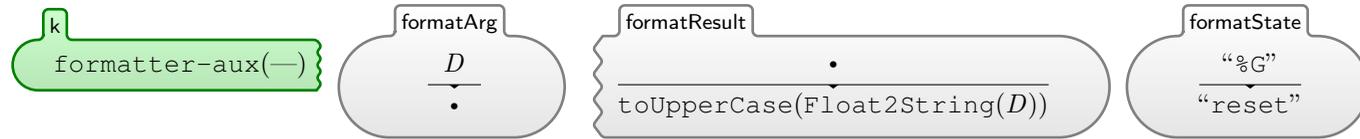
A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

RULE **FORMAT-%G-START**RULE **FORMAT-%G**

RULE **FORMAT-%G-START**



RULE **FORMAT-%G**

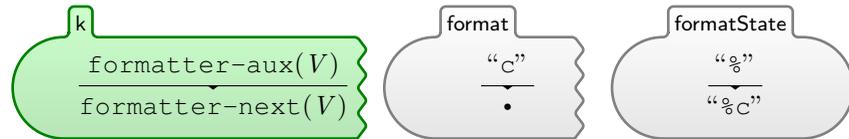


(n1570) §7.21.6.1 ¶8

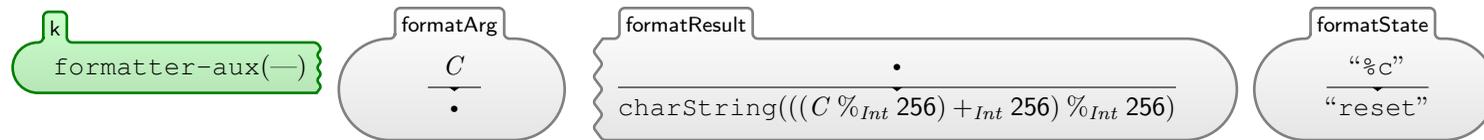
c If no `l` length modifier is present, the **int** argument is converted to an **unsigned char**, and the resulting character is written.

If an `l` length modifier is present, the `wint_t` argument is converted as if by an `ls` conversion specification with no precision and an argument that points to the initial element of a two-element array of `wchar_t`, the first element containing the `wint_t` argument to the `lc` conversion specification and the second a null wide character.

RULE **FORMAT-%C-START**



RULE **FORMAT-%C**

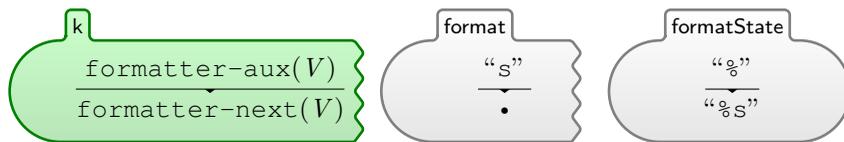


(n1570) §7.21.6.1 ¶8

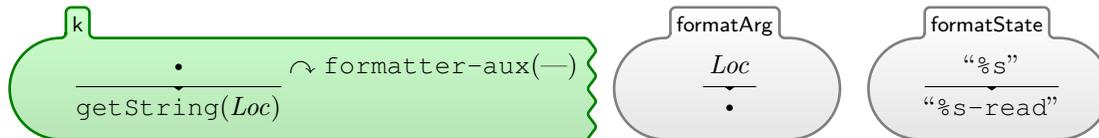
s If no `l` length modifier is present, the argument shall be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of `wchar_t` type. Wide characters from the array are converted to multibyte characters (each as if by a call to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.

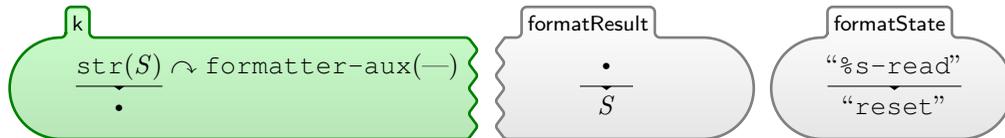
RULE FORMAT-%S-START



RULE FORMAT-%S



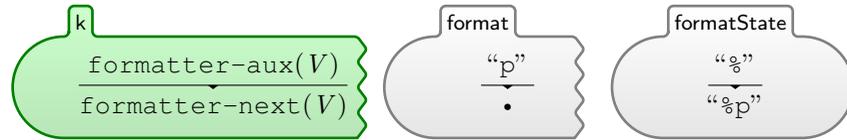
RULE FORMAT-%S-DONE



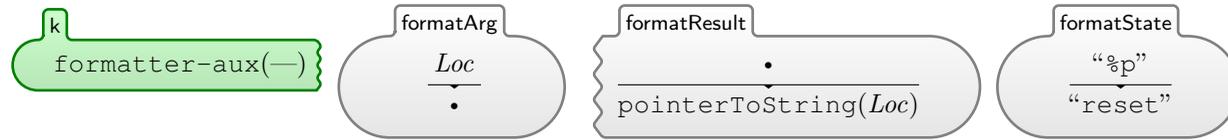
(n1570) §7.21.6.1 ¶8

p The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

RULE **FORMAT-%P-START**



RULE **FORMAT-%P**



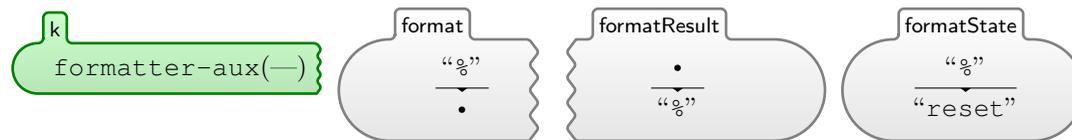
(n1570) §7.21.6.1 ¶8

n The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to `fprintf`. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

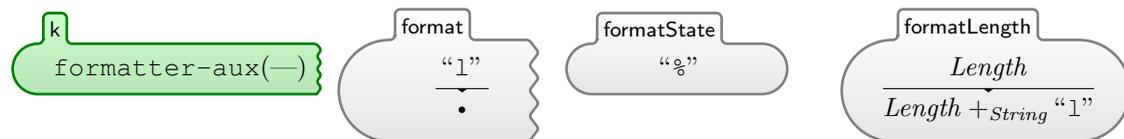
(n1570) §7.21.6.1 ¶8

% A `%` character is written. No argument is converted. The complete conversion specification shall be `%%`.

RULE **FORMAT-%%**



RULE **FORMAT-%L**



END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-STDLIB

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

RULE DEBUG

$$\frac{k \quad \text{prepareBuiltin(Identifier("__debug"), \text{---})}}{\text{skipval}}$$

[interpRule]

RULE DEBUG-K

$$\frac{k \quad \text{debug}}{\bullet}$$

[interpRule]

RULE DEBUG-M

$$\frac{k \quad \text{debug-m(\text{---})}}{\bullet}$$

[interpRule]

RULE EXIT

$$\frac{k \quad \text{prepareBuiltin(Identifier("exit"), I:t(\text{---}, \text{int}))} \curvearrow \text{---}}{I:t(\bullet, \text{int})}$$

RULE ABORT

$$\frac{k \quad \text{prepareBuiltin(Identifier("abort"), \bullet) \curvearrow \text{---}}{\text{printString("Aborted")} \curvearrow 134:t(\bullet, \text{int})}$$

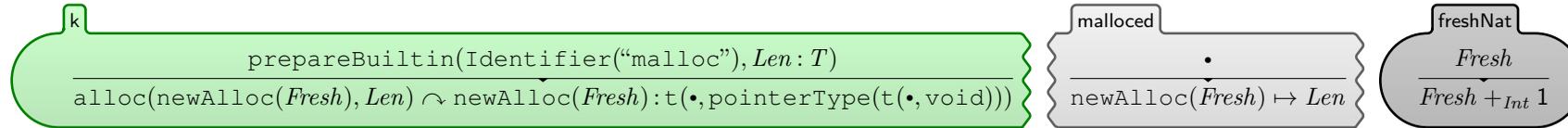
(n1570) §7.22.3.4 ¶2–3 The `malloc` function allocates space for an object whose size is specified by `size` and whose value is indeterminate. The `malloc` function returns either a null pointer or a pointer to the allocated space.

SYNTAX $K ::= \text{newAlloc}(Nat)$

DEFINE

$$\frac{\text{newAlloc}(Fresh)}{\text{loc}(\text{threadId}(\text{allocatedDuration}) +_{Int} Fresh, 0, 0)}$$

RULE **MALLOC**

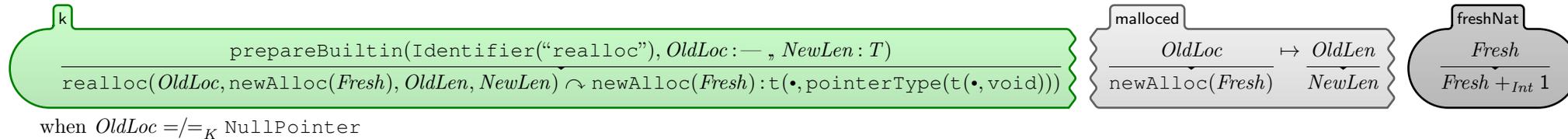


(n1570) §7.22.3.5 ¶2–4 The `realloc` function deallocates the old object pointed to by `ptr` and returns a pointer to a new object that has the size specified by `size`. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

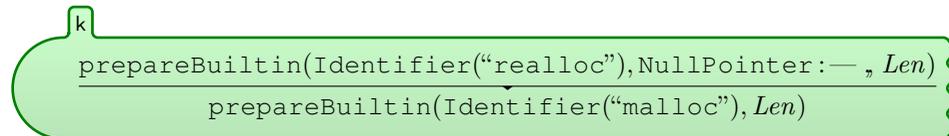
If `ptr` is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to the `free` or `realloc` function, the behavior is undefined. If memory for the new object cannot be allocated, the old object is not deallocated and its value is unchanged.

The `realloc` function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object could not be allocated.

RULE **REALLOC**



RULE **REALLOC-NULL**



SYNTAX $K ::= \text{calloc-aux}$

RULE CALLOC

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"calloc"}), N : -, Size : -)}{\text{prepareBuiltin}(\text{Identifier}(\text{"malloc"}), N *_{Int} Size : \text{cfg:sizeut}) \rightsquigarrow \text{calloc-aux}}$$

RULE CALLOC-AUX

$$\frac{\bullet \rightsquigarrow Loc : t(-, \text{pointerType}(t(-, \text{void}))) \rightsquigarrow \text{calloc-aux}}{\text{zeroBlock}(Loc)}$$

RULE FREE

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"free"}), Loc : t(-, \text{pointerType}(-)))}{\text{deleteSizedBlock}(Loc, Len) \rightsquigarrow \text{skipval}}$$

$$\frac{\text{malloced} \quad Loc \mapsto Len}{\bullet}$$

RULE RAND

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"rand"}), \bullet)}{(\text{absInt randomRandom}(Fresh)) \%_{Int} \max(t(\bullet, \text{int})) : t(\bullet, \text{int})}$$

$$\frac{\text{randNat} \quad Fresh}{Fresh +_{Int} 1}$$

RULE SRAND

$$\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"srand"}), N : t(\bullet, \text{unsigned-int}))}{\text{skipval}}$$

$$\frac{\text{randNat}}{N}$$

END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-STRING

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

SYNTAX $K ::= \text{strcpy}(K, K, K)$

RULE STRCPY-START

$$\frac{\text{prepareBuiltin(Identifier("strcpy"), Dest:t(-, pointerType(-)), Src:t(-, pointerType(-)))}}{\text{strcpy(Dest, Src, Dest)}}$$

RULE STRCPY-PRE

$$\frac{\bullet \quad \sim \text{strcpy}(-, \text{Src}, -)}{\text{read}(Src, t(\bullet, \text{char})) \quad \text{Src} +_{Int} 1}$$

RULE STRCPY-SOME

$$\frac{I:T \quad \sim \text{strcpy}(\text{Dest}, -, -)}{\text{write}(lv(\text{Dest}, t(\bullet, \text{char})), I:T) \quad \text{Dest} +_{Int} 1}$$

when $I \neq_{Int} 0$

RULE STRCPY-DONE

$$\frac{0:T \quad \sim \text{strcpy}(\text{Dest}, -, \text{Orig})}{\text{write}(lv(\text{Dest}, t(\bullet, \text{char})), 0:T) \quad \sim \text{Orig}:t(\bullet, \text{pointerType}(t(\bullet, \text{char})))}$$

END MODULE

MODULE DYNAMIC-C-STANDARD-LIBRARY-THREADS

IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

SYNTAX $K ::= \text{spawn-aux}(Nat, Value, Value)$
| $\text{join-aux}(Nat, Value)$

SYNTAX $Nat ::= \text{thrd-busy}$
| thrd-error
| thrd-nomem
| thrd-success
| thrd-timeout

SYNTAX $K ::= \text{threadClosed}$

SYNTAX $Nat ::= \text{threadId}(Nat, Nat)$

SYNTAX $K ::= \text{threadJoining}(Nat)$
| threadRunning

MACRO
 $\text{thrd-success} = 0 : t(\bullet, \text{int})$

MACRO
 $\text{thrd-error} = 1 : t(\bullet, \text{int})$

MACRO
 $\text{thrd-timeout} = 2 : t(\bullet, \text{int})$

MACRO
 $\text{thrd-busy} = 3 : t(\bullet, \text{int})$

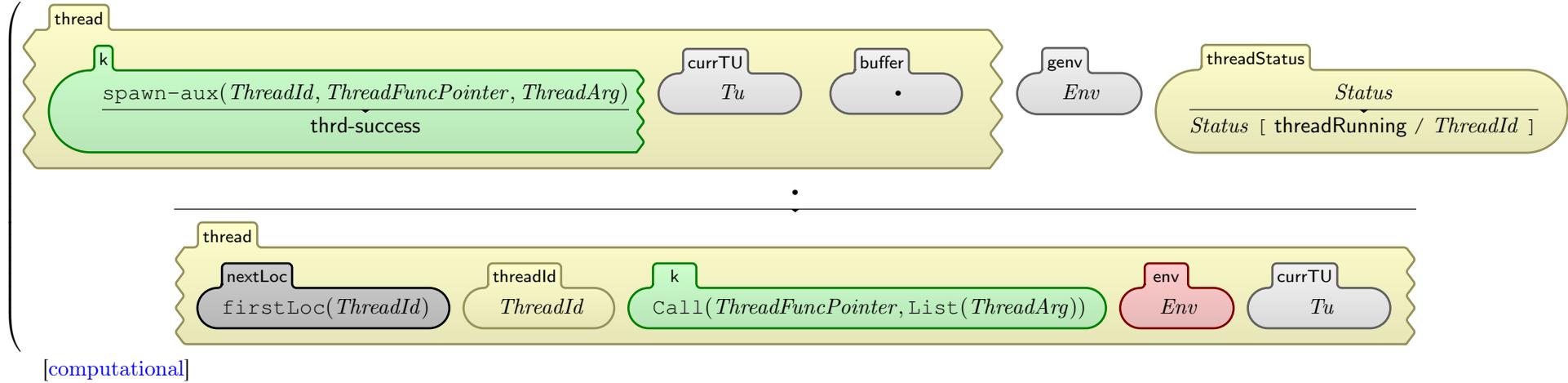
MACRO
 $\text{thrd-nomem} = 4 : t(\bullet, \text{int})$

RULE **THR-CREATE-START**

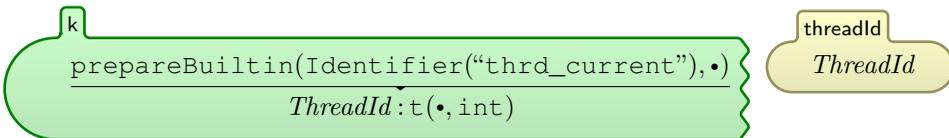
k
 $\frac{\text{prepareBuiltin}(\text{Identifier}(\text{"thrd_create"}), \text{ThreadIdPointer}, \text{ThreadFuncPointer}, \text{ThreadArg})}{(* \text{ThreadIdPointer}) := \text{Fresh} : t(\bullet, \text{int}); \curvearrowright \text{spawn-aux}(\text{Fresh}, \text{ThreadFuncPointer}, \text{ThreadArg})}$

nextThreadId
 $\frac{\text{Fresh}}{\text{Fresh} +_{\text{Int}} 1}$

RULE THRD-CREATE



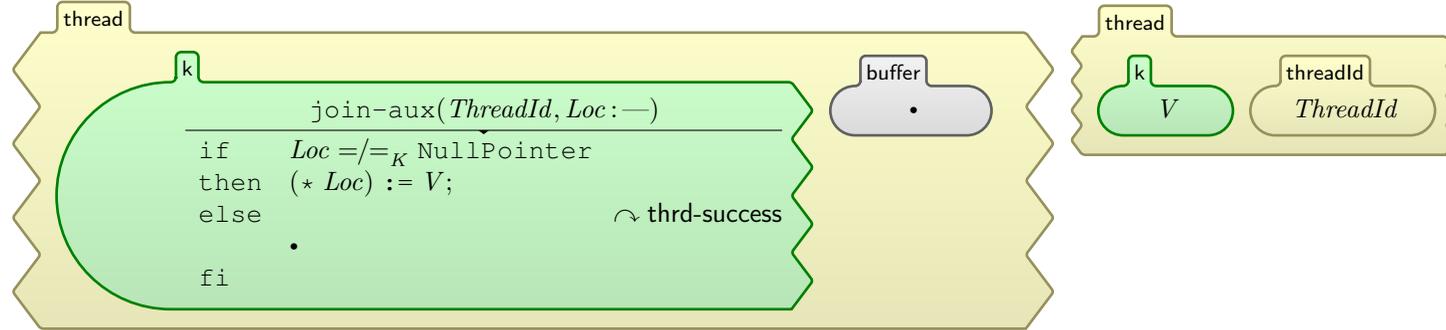
RULE THRD-CURRENT



RULE THRD-JOIN-START

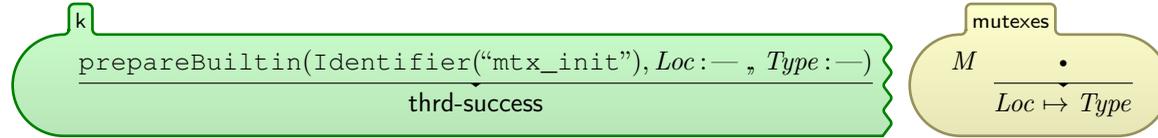


RULE THRD-JOIN



[computational]

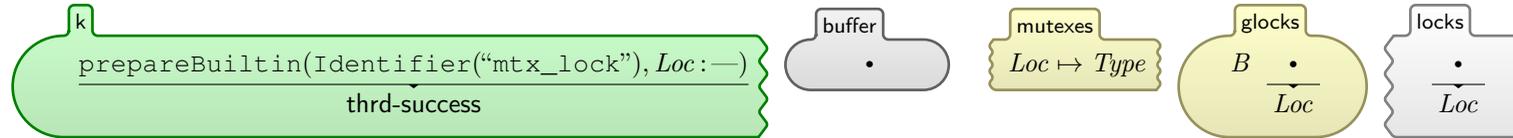
RULE MTX-INIT



when $(\neg_{Bool} (Loc \text{ in } (\text{keys } M))) \wedge_{Bool} (Type ==_{Int} \text{cfg:mtxPlain})$

342

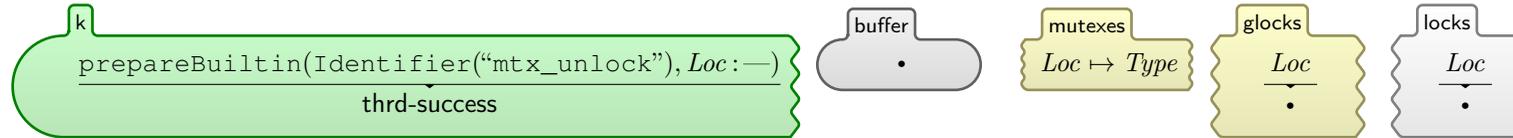
RULE MTX-LOCK



when $(\neg_{Bool} (Loc \text{ in } B)) \wedge_{Bool} (Type ==_{Int} \text{cfg:mtxPlain})$

[computational]

RULE MTX-UNLOCK



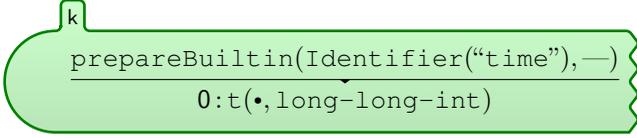
when $Type ==_{Int} \text{cfg:mtxPlain}$

[computational]

END MODULE

```
MODULE DYNAMIC-C-STANDARD-LIBRARY-TIME

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

  RULE TIME
    
  END MODULE
```

```
MODULE DYNAMIC-C-STANDARD-LIBRARY-MISC

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-INCLUDE

  END MODULE
```

```
MODULE DYNAMIC-C-STANDARD-LIBRARY

  IMPORTS DYNAMIC-INCLUDE

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-HELPERS

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-MATH

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-SETJMP

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-STDARG

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-STDDEF

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-STDIO

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-STDLIB

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-STRING

  IMPORTS DYNAMIC-C-STANDARD-LIBRARY-THREADS
```

```
IMPORTS DYNAMIC-C-STANDARD-LIBRARY-TIME
```

```
IMPORTS DYNAMIC-C-STANDARD-LIBRARY-MISC
```

```
END MODULE
```

A.9 Error Handling

This section is a collection of rules that discern precise error conditions and generates English error messages in those cases. While these rules do not affect the correctness of the semantics, they make working with the semantics much easier for the end user.

```
MODULE DYNAMIC-SEMANTICS-ERRORS-INCLUDE
```

```
  IMPORTS COMMON-C-SEMANTICS
```

```
  IMPORTS DYNAMIC-C-SEMANTICS-MISC
```

```
  IMPORTS DYNAMIC-C-EXPRESSIONS
```

```
  IMPORTS DYNAMIC-C-TYPING
```

```
  IMPORTS DYNAMIC-C-DECLARATIONS
```

```
  IMPORTS DYNAMIC-C-MEMORY
```

```
  IMPORTS DYNAMIC-C-STATEMENTS
```

```
  IMPORTS DYNAMIC-C-CONVERSIONS
```

```
  IMPORTS DYNAMIC-C-STANDARD-LIBRARY
```

```
END MODULE
```

```
MODULE DYNAMIC-C-ERRORS
```

```
  IMPORTS DYNAMIC-SEMANTICS-ERRORS-INCLUDE
```

```
  SYNTAX  $K ::= \text{Error}(String, String)$  [function]
```

```
  DEFINE
```

$$\text{Error}(Name, Msg)$$

$((("Error:" + String Name) + String "\n") + String "Description:") + String Msg$

```
  SYNTAX  $K ::= \text{ICE}(String, String)$  [function]
```

```
  DEFINE
```

$$\text{ICE}(Name, Msg)$$

$\text{Error}(Name, Msg) + String "\nNOTE: Please send a test case exhibiting this bug to celliso2@illinois.edu; it could indicate an internal error in KCC."$

```
  SYNTAX  $Bag ::= \text{halt } Bag$  [function]
```

DEFINE
 $\text{halt } \langle \frac{k}{\text{halted-k}} \rangle \text{---} \langle / \frac{k}{\text{halted-k}} \rangle$

DEFINE
 $\frac{\text{halt } \langle L \rangle K \langle / L \rangle}{\langle L \rangle K \langle / L \rangle}$
 when $L \neq_{\text{CellLabel}} k$

DEFINE
 $\frac{\text{halt } \langle L \rangle B \langle / L \rangle}{\langle L \rangle \text{halt } B \langle / L \rangle}$

DEFINE
 $\frac{\text{halt } \langle L \rangle K \langle / L \rangle}{\langle L \rangle K \langle / L \rangle}$

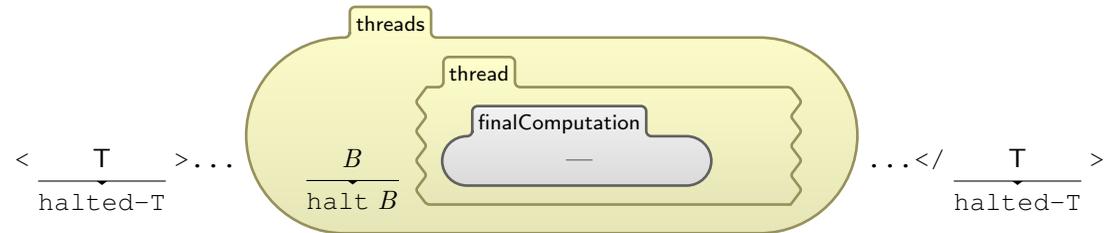
DEFINE
 $\frac{\text{halt } \langle L \rangle K \langle / L \rangle}{\langle L \rangle K \langle / L \rangle}$

DEFINE
 $\frac{\text{halt } \langle L \rangle K \langle / L \rangle}{\langle L \rangle K \langle / L \rangle}$

DEFINE
 $\frac{\text{halt } \bullet}{\bullet}$

RULE **HALT-START**

347



RULE **ERR00001**

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{cast}(t(-, T), \text{emptyValue}) \dots \langle \frac{k}{\text{finalComputation}} \rangle \\ \hline \text{errorCell} \\ \text{Error}(\text{"00001"}, \text{"Casting empty value to type other than void."}) \end{array} \right)$$

when $T \neq_K \text{void}$

RULE **ERR00002**

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{assert}(\text{false}, 2) \dots \langle \frac{k}{\text{finalComputation}} \rangle \\ \hline \text{errorCell} \\ \text{Error}(\text{"00002"}, \text{"Reading outside the bounds of an object."}) \end{array} \right)$$

RULE **ERR00003**

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{assert}(\text{false}, 3) \dots \langle \frac{k}{\text{finalComputation}} \rangle \\ \hline \text{errorCell} \\ \text{Error}(\text{"00003"}, \text{"Unsequenced side effect on scalar object with value computation of same object."}) \end{array} \right)$$

RULE **ERR00005**

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{extractByteFromMem}(\text{loc}(Block, -, -)) \dots \langle \frac{k}{\text{finalComputation}} \rangle \text{memory } M \\ \hline \text{errorCell} \\ \text{ICE}(\text{"00005"}, \text{"Referring to an object outside of its lifetime."}) \end{array} \right)$$

when $\neg_{Bool}(Block \text{ in } \text{gatherInnerCells}(M, \text{basePtr}))$

RULE ERR00006

$$\left(\begin{array}{c} \frac{\langle \text{finalComputation} \rangle}{k} \text{ joinIntegerBytes-aux}(T, -, \text{piece}(\text{unknown}(Len), Len), -) \dots \langle / \text{finalComputation} \rangle \\ \hline \text{errorCell} \\ \text{Error}(\text{"00006"}, \text{"Reading unspecified (possibly uninitialized) memory, or trying to read a pointer or float through an integer type."}) \end{array} \right)$$

when $\neg_{Bool} \text{isCharType}(T)$

RULE ERR00007

$$\left(\begin{array}{c} \frac{\langle \text{finalComputation} \rangle}{k} \text{ checkValidLoc-aux}(\text{loc}(Block, -, -)) \dots \langle / \text{finalComputation} \rangle \text{ memory } M \\ \hline \text{errorCell} \\ \text{Error}(\text{"00007"}, \text{"Referring to an object outside of its lifetime."}) \end{array} \right)$$

when $\neg_{Bool} (Block \text{ in } \text{gatherInnerCells}(M, \text{basePtr}))$

RULE ERR00008

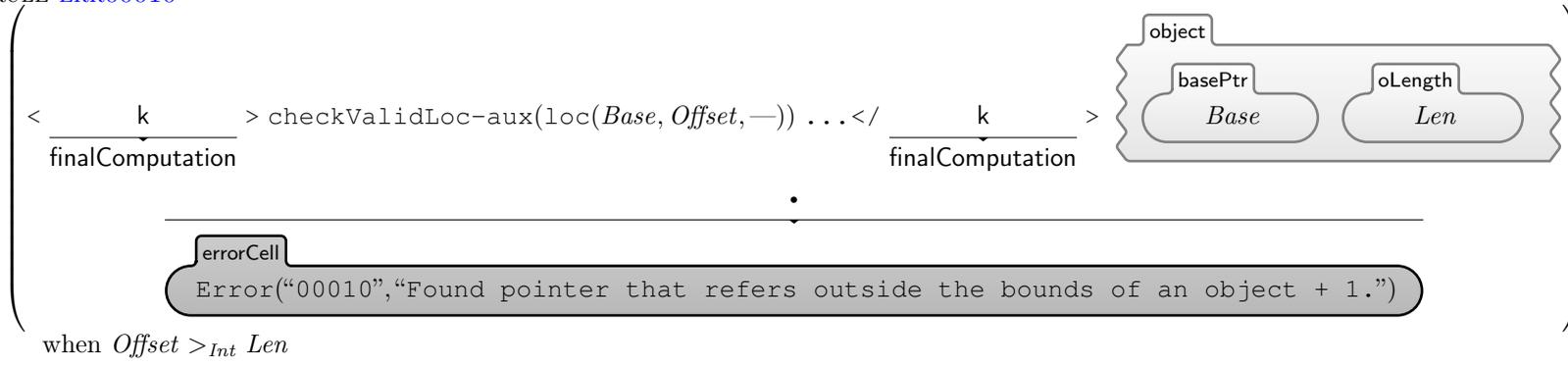
$$\left(\begin{array}{c} \frac{\langle \text{finalComputation} \rangle}{k} \text{ concretize}(t(-, \text{pointerType}(-)), \text{dataList}(\text{piece}(\text{unknown}(Len), Len), -, -)) \dots \langle / \text{finalComputation} \rangle \\ \hline \text{errorCell} \\ \text{Error}(\text{"00008"}, \text{"Reading uninitialized memory."}) \end{array} \right)$$

RULE ERR00009

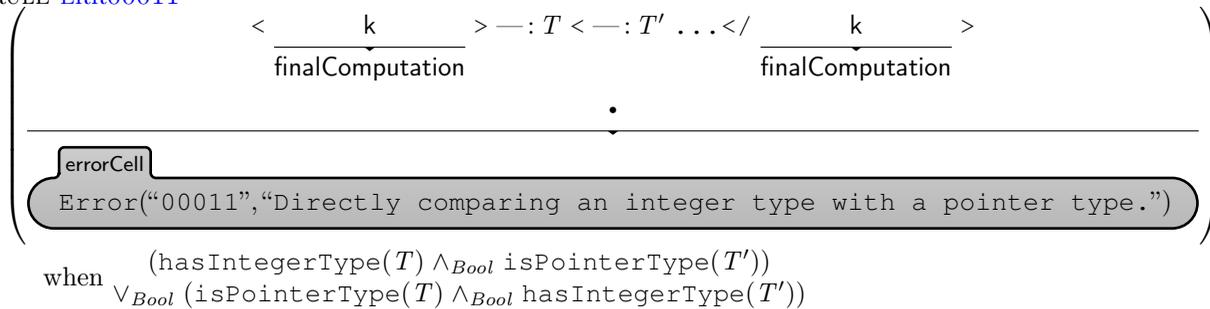
$$\left(\begin{array}{c} \frac{\langle \text{finalComputation} \rangle}{k} \text{ concretize}(T, \text{dataList}(\text{piece}(\text{unknown}(Len), Len), -, -)) \dots \langle / \text{finalComputation} \rangle \\ \hline \text{errorCell} \\ \text{Error}(\text{"00009"}, \text{"Reading uninitialized memory."}) \end{array} \right)$$

when $\text{isFloatType}(T)$

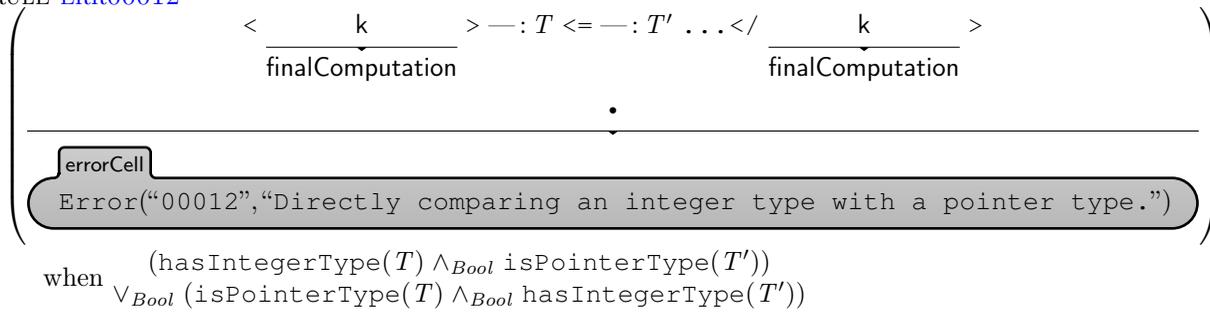
RULE ERR00010



RULE ERR00011



RULE ERR00012



RULE ERR00013

$$\left(\frac{\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{---} : T \text{---} : T' \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error("00013", "Directly comparing an integer type with a pointer type.")} \end{array}}{\text{---}} \right)$$

when $(\text{hasIntegerType}(T) \wedge_{Bool} \text{isPointerType}(T'))$
 $\vee_{Bool} (\text{isPointerType}(T) \wedge_{Bool} \text{hasIntegerType}(T'))$

RULE ERR00014

$$\left(\frac{\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{---} : T \text{---} : T' \text{---} : T' \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error("00014", "Directly comparing an integer type with a pointer type.")} \end{array}}{\text{---}} \right)$$

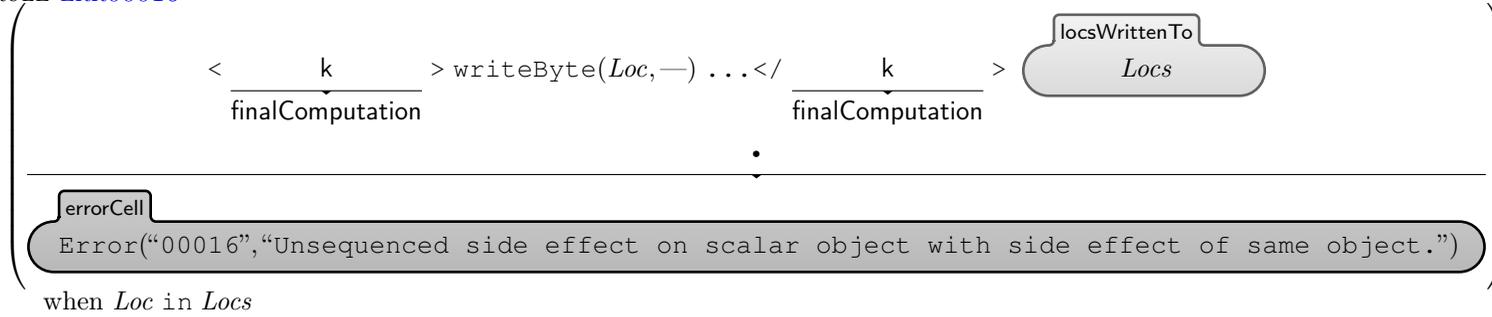
when $(\text{hasIntegerType}(T) \wedge_{Bool} \text{isPointerType}(T'))$
 $\vee_{Bool} (\text{isPointerType}(T) \wedge_{Bool} \text{hasIntegerType}(T'))$

RULE ERR00015

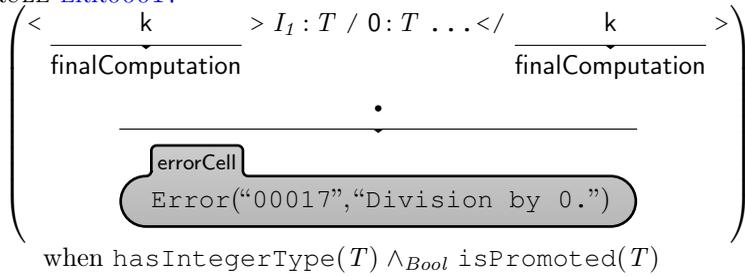
$$\left(\frac{\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{arithInterpret}(T, I) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error("00015", "Signed overflow.")} \end{array}}{\text{---}} \right)$$

when $\text{hasSignedIntegerType}(T) \wedge_{Bool} (\neg_{Bool} ((\min(T) \leq_{Int} I) \wedge_{Bool} (\max(T) \geq_{Int} I)))$

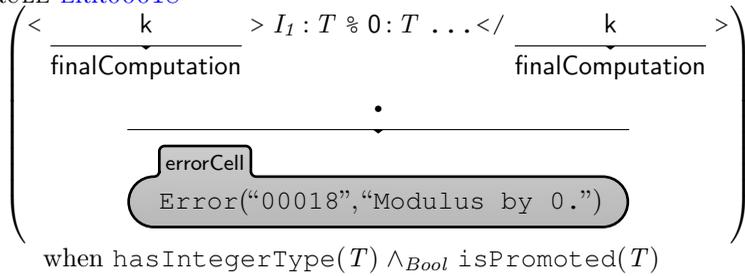
RULE ERR00016



RULE ERR00017



RULE ERR00018



RULE [ERR00019](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle I_1 : T \% I_2 : T \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00019"}, \text{"Signed overflow."}) \end{array} \right)$$

when $((\text{hasIntegerType}(T) \wedge_{Bool} (\neg_{Bool} ((\min(T) \leq_{Int} (I_1 \div_{Int} I_2)) \wedge_{Bool} (\max(T) \geq_{Int} (I_1 \div_{Int} I_2)))))) \wedge_{Bool} \text{isPromoted}(T) \wedge_{Bool} (I_2 \neq_{Int} 0)$

RULE [ERR00020](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{writeByte}(\text{loc}(\text{Base}, \text{Offset}, -), -) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00020"}, \text{"Tried to write outside the bounds of an object."}) \end{array} \right)$$

when $\neg_{Bool} (\text{Offset} <_{Int} \text{Len})$

The diagram shows an object structure with fields *basePtr* (containing *Base*) and *oLength* (containing *Len*).

RULE [ERR00021](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{Identifier}(S) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00021"}, (\text{"Trying to look up identifier"} + \text{String } S) + \text{String } \text{" , but no such identifier is in scope."}) \end{array} \right)$$

when $\neg_{Bool} \$\text{hasMapping}(M, \text{Identifier}(S))$

The diagram shows an environment *env* containing a mapping *M*.

RULE [ERR00022](#)

$$\left(\frac{\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{leftShiftInterpret}(T, I, E_1 : T) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \vdots \end{array}}{\text{errorCell} \\ \text{Error}(\text{"00022"}, \text{"Trying to left-shift a negative signed value."})} \right)$$

when $\text{hasSignedIntegerType}(T) \wedge_{Bool} (E_1 <_{Int} 0)$

RULE [ERR00023](#)

$$\left(\frac{\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{leftShiftInterpret}(T, I, E_1 : T) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \vdots \end{array}}{\text{errorCell} \\ \text{Error}(\text{"00023"}, \text{"Trying to left-shift a signed value, but the result is not representable in the result type."})} \right)$$

when $\text{hasSignedIntegerType}(T) \wedge_{Bool} (\neg_{Bool} (I \leq_{Int} (2 \wedge_{Int} (\text{absInt numBits}(T))))))$

RULE [ERR00024](#)

$$\left(\frac{\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{arithInterpret}(\neg, I \&_{Int} I') \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \vdots \end{array}}{\text{errorCell} \\ \text{Error}(\text{"00024"}, \text{"Bitwise \& used on a symbolic number (address) or float."})} \right)$$

when $(\neg_{Bool} \text{isConcreteNumber}(I)) \vee_{Bool} (\neg_{Bool} \text{isConcreteNumber}(I'))$

RULE [ERR00025](#)

$$\left(\frac{\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{callMain-aux}(t(\neg, \text{functionType}(t(\neg, T), \neg)), \neg, \neg, \neg) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \vdots \\ \text{errorCell} \\ \text{Error}(\text{"00025"}, \text{"Main must return an int."}) \end{array}}{\text{when } T \neq_K \text{ int}}$$

RULE ERR00026

$$\left(\frac{\begin{array}{c} \text{finalComputation} \\ \vdots \\ \text{Error}(\text{"00026", "If main has arguments, the type of the first argument must be equivalent to \"int\."}) \end{array}}{\text{when } (T \neq_K \text{ int}) \wedge_{Bool} (T \neq_K \text{ void})} \frac{\langle \frac{k}{\text{finalComputation}} \rangle \text{ callMain-aux}(t(-, \text{functionType}(t(-, \text{int}), \text{typedDecl}(t(-, T), -), -)), -, -, -) \dots \langle / \frac{k}{\text{finalComputation}} \rangle}{\text{finalComputation}} \right)$$

SYNTAX $Bool ::= \text{isArgvType}(Type)$ [function]

DEFINE

$$\frac{\text{isArgvType}(t(-, T))}{\text{false}} \text{ when } \neg_{Bool} \left(\begin{array}{c} ((\text{getKLabel}(T)) ==_{KLabel} \text{incompleteArrayType}) \\ \vee_{Bool} ((\text{getKLabel}(T)) ==_{KLabel} \text{incompleteArrayType}) \end{array} \right)$$

DEFINE

$$\frac{\text{isArgvType}(t(-, \text{incompleteArrayType}(t(-, T))))}{\text{false}} \text{ when } \neg_{Bool} ((\text{getKLabel}(T)) ==_{KLabel} \text{pointerType})$$

DEFINE

$$\frac{\text{isArgvType}(t(-, \text{incompleteArrayType}(t(-, \text{pointerType}(t(-, T))))))}{\text{false}} \text{ when } T \neq_K \text{ char}$$

DEFINE

$$\frac{\text{isArgvType}(t(-, \text{pointerType}(t(-, T))))}{\text{false}} \text{ when } \neg_{Bool} ((\text{getKLabel}(T)) ==_{KLabel} \text{pointerType})$$

DEFINE

$$\frac{\text{isArgvType}(t(-, \text{pointerType}(t(-, \text{pointerType}(t(-, T))))))}{\text{false}} \text{ when } T \neq_K \text{ char}$$

RULE ERR00027

$$\left(\begin{array}{c} \frac{\langle \frac{k}{\text{finalComputation}} \rangle \text{callMain-aux}(t(-, \text{functionType}(t(-, \text{int}), \text{typedDecl}(t(-, \text{int}), -), \text{typedDecl}(T, -))), -, -, -) \dots \langle / \frac{k}{\text{finalComputation}} \rangle}{\text{finalComputation}} \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00027"}, \text{"If main has arguments, the type of the second argument must be equivalent to char** ."}) \end{array} \right)$$

when $\neg_{Bool} \text{isArgvType}(T)$

RULE ERR00028

$$\left(\begin{array}{c} \frac{\langle \frac{k}{\text{finalComputation}} \rangle \text{callMain-aux}(t(-, \text{functionType}(t(-, \text{int}), -, -, -, -)), -, -, -) \dots \langle / \frac{k}{\text{finalComputation}} \rangle}{\text{finalComputation}} \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00028"}, \text{"Main can only have zero or two arguments."}) \end{array} \right)$$

RULE ERR00029

$$\left(\begin{array}{c} \frac{\langle \frac{k}{\text{finalComputation}} \rangle \text{callMain-aux}(t(-, \text{functionType}(t(-, \text{int}), \text{typedDecl}(t(-, T), -))), -, -, -) \dots \langle / \frac{k}{\text{finalComputation}} \rangle}{\text{finalComputation}} \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00029"}, \text{"Main can only have zero or two arguments."}) \end{array} \right)$$

when $T \neq_K \text{void}$

RULE ERR00030

$$\left(\begin{array}{c} \frac{\langle \frac{k}{\text{finalComputation}} \rangle \text{loc}(Base, -, -): T < \text{loc}(Base', -, -): T \dots \langle / \frac{k}{\text{finalComputation}} \rangle}{\text{finalComputation}} \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00030"}, \text{"Cannot compare pointers with different base objects using <."}) \end{array} \right)$$

when $Base \neq_K Base'$

RULE [ERR00031](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{loc}(Base, -, -): T > \text{loc}(Base', -, -): T \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00031"}, \text{"Cannot compare pointers with different base objects using >."}) \end{array} \right)$$

when $Base \neq_K Base'$

RULE [ERR00032](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{loc}(Base, -, -): T \leq \text{loc}(Base', -, -): T \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00032"}, \text{"Cannot compare pointers with different base objects using <=."}) \end{array} \right)$$

when $Base \neq_K Base'$

RULE [ERR00033](#)

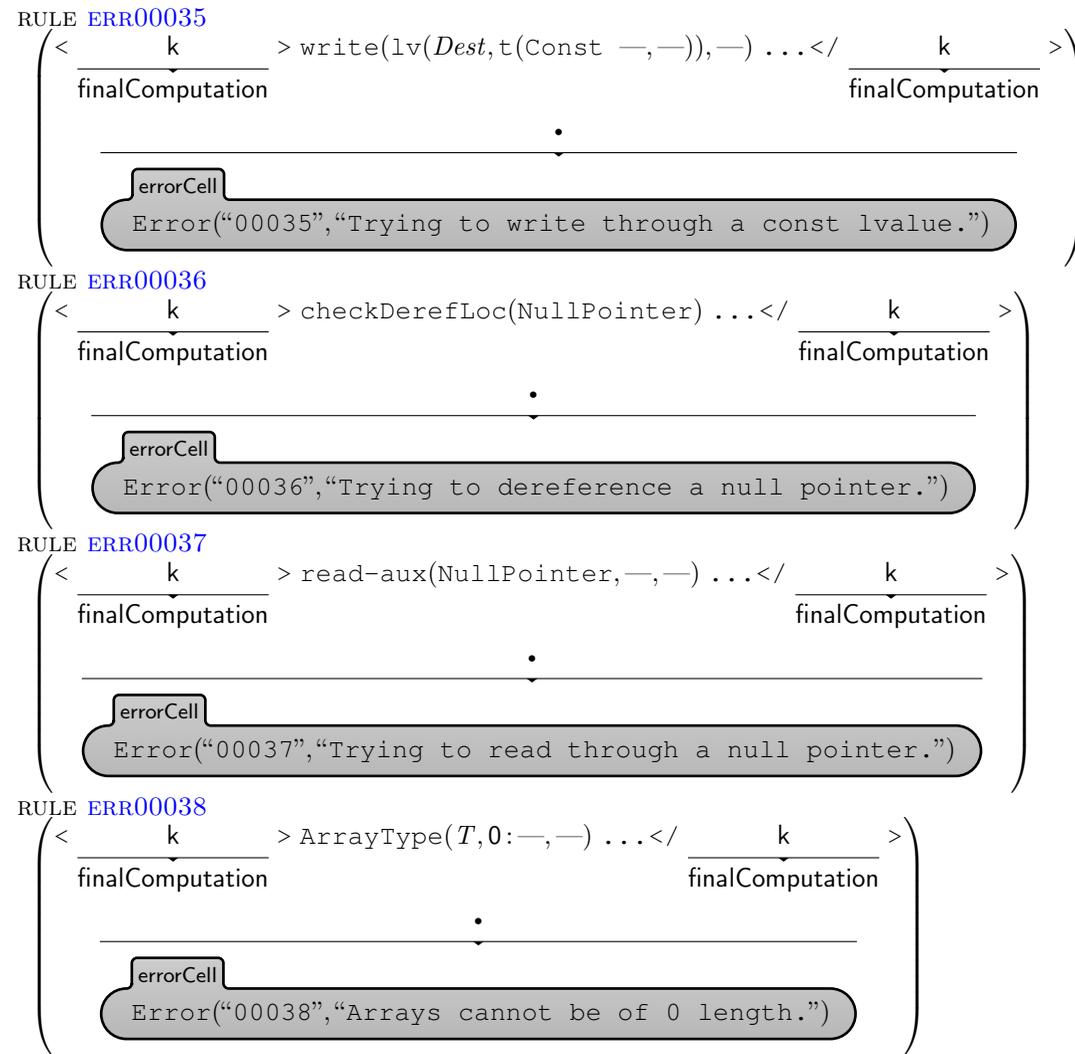
$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{loc}(Base, -, -): T \geq \text{loc}(Base', -, -): T \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00033"}, \text{"Cannot compare pointers with different base objects using >=."}) \end{array} \right)$$

when $Base \neq_K Base'$

RULE [ERR00034](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{cast}(t(-, T), \text{skipval}) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00034"}, \text{"Casting void type to non-void type."}) \end{array} \right)$$

when $T \neq_K \text{void}$



RULE [ERR00039](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{addUnion}(S, \bullet) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00039"}, \text{"Unions cannot be empty."}) \end{array} \right)$$

RULE [ERR00040](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{addStruct}(S, \bullet) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00040"}, \text{"Structs cannot be empty."}) \end{array} \right)$$

RULE [ERR00041](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{types}(t(-, \text{void}), t(-, T)) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00041"}, \text{"If one of a conditional expressions branches has void type, the other must also have void type."}) \end{array} \right)$$

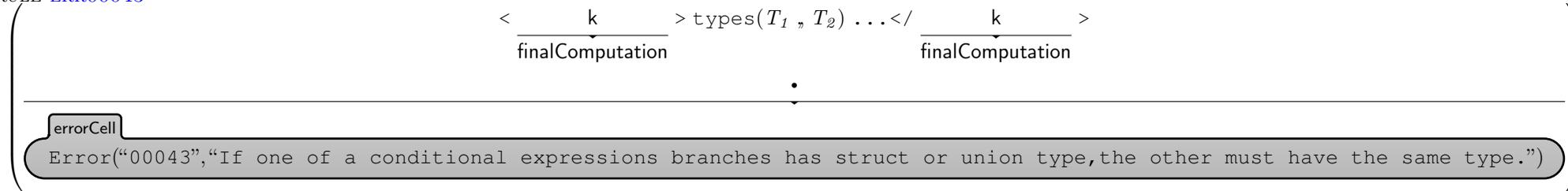
when $T \neq_K \text{void}$

RULE [ERR00042](#)

$$\left(\begin{array}{c} \langle \frac{k}{\text{finalComputation}} \rangle \text{types}(t(-, T), t(-, \text{void})) \dots \langle / \frac{k}{\text{finalComputation}} \rangle \\ \cdot \\ \text{errorCell} \\ \text{Error}(\text{"00042"}, \text{"If one of a conditional expressions branches has void type, the other must also have void type."}) \end{array} \right)$$

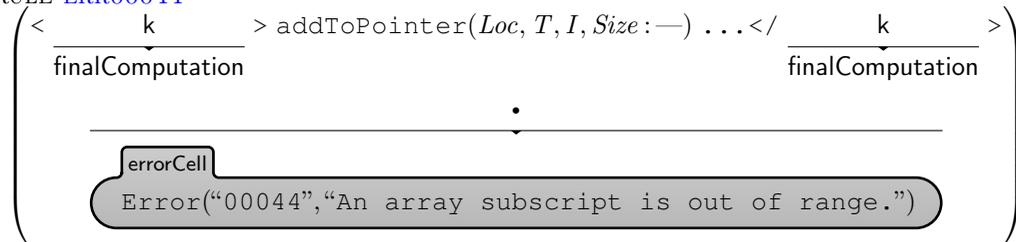
when $T \neq_K \text{void}$

RULE ERR00043



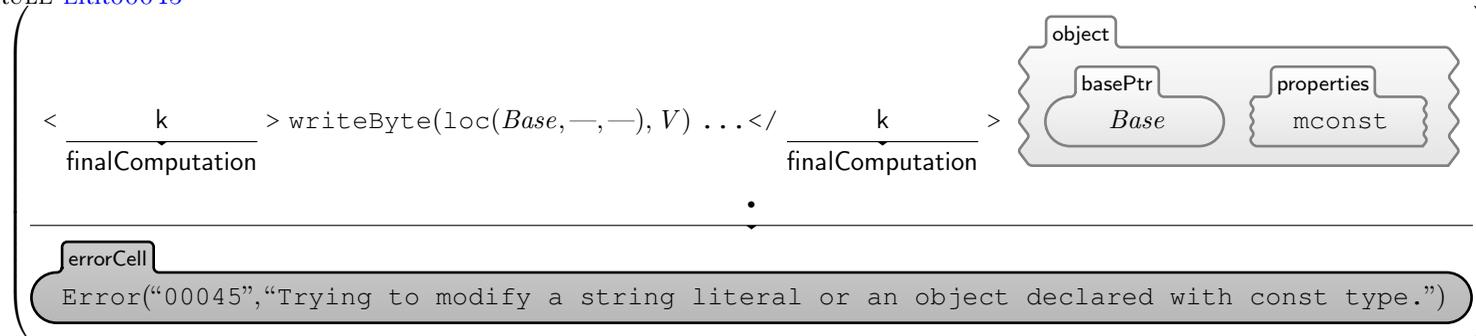
when $\left((T_1 \neq_K T_2) \wedge_{Bool} \left(\frac{\text{isStructType}(T_1)}{\vee_{Bool} \text{isUnionType}(T_1)} \right) \right) \wedge_{Bool} \left(\frac{\text{isStructType}(T_2)}{\vee_{Bool} \text{isUnionType}(T_2)} \right)$

RULE ERR00044

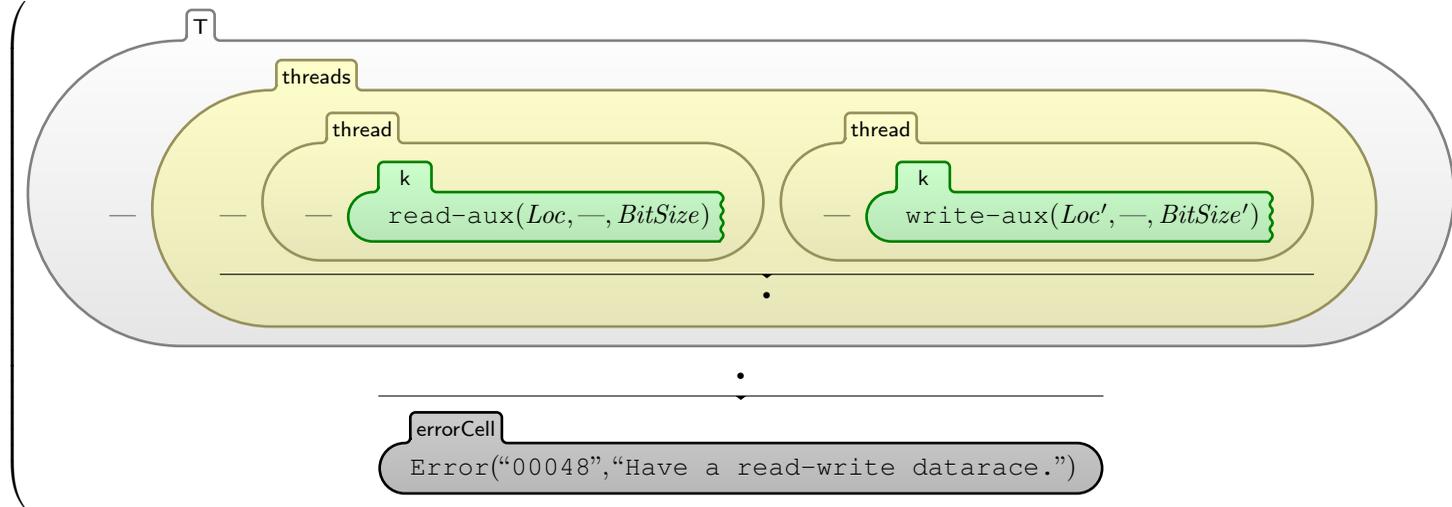


when $\neg_{Bool} \text{ifFromArrayInBounds}(T, I)$

RULE ERR00045

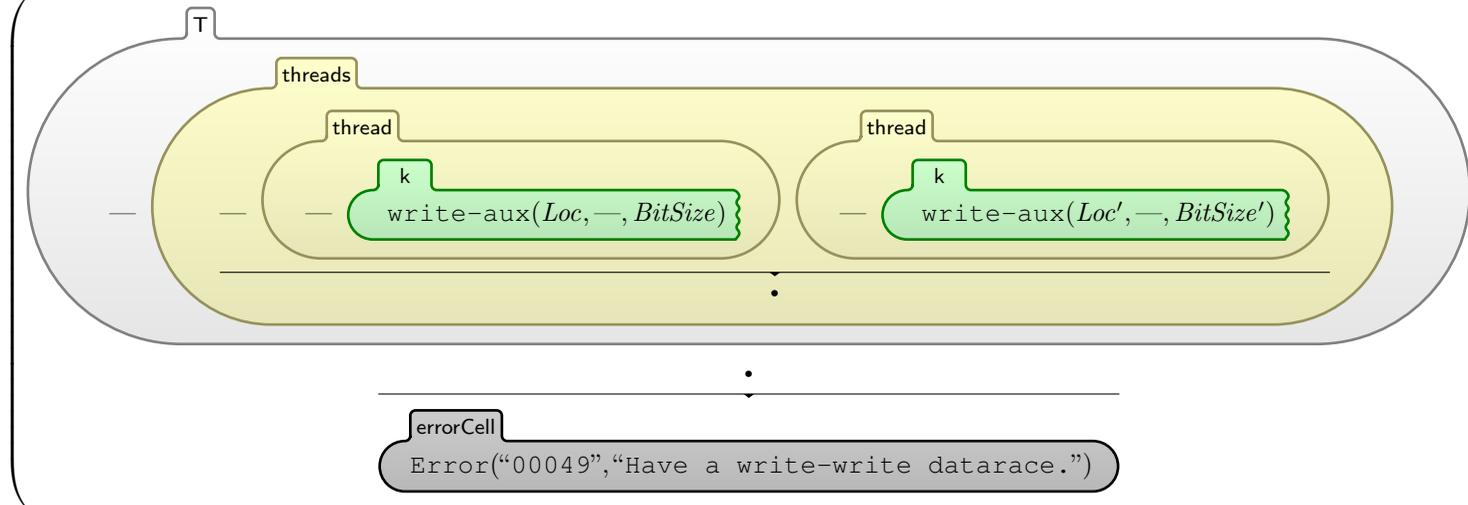


RULE READ-WRITE-RACE



when $((Loc \leq_{Int} Loc') \wedge_{Bool} (Loc' <_{Int} (Loc +_{Int} \text{bitsToBytes}(BitSize))))$
 $\vee_{Bool} ((Loc >_{Int} Loc') \wedge_{Bool} ((Loc' +_{Int} \text{bitsToBytes}(BitSize')) >_{Int} Loc))$
 [computational]

RULE WRITE-WRITE-RACE



when $((Loc \leq_{Int} Loc') \wedge_{Bool} (Loc' <_{Int} (Loc +_{Int} \text{bitsToBytes}(BitSize))))$
 $\vee_{Bool} ((Loc >_{Int} Loc') \wedge_{Bool} ((Loc' +_{Int} \text{bitsToBytes}(BitSize')) >_{Int} Loc))$
[computational]

END MODULE

A.10 Miscellaneous

This section is a collection of miscellaneous rules and syntax for all of the helper operators used elsewhere in the semantics. Some of these operators may be useful for other semantics defined in \mathbb{K} , but have not yet made it in the \mathbb{K} library. This section also contains descriptions of values.

MODULE COMMON-INCOMING-MODULES

IMPORTS C-SYNTAX

IMPORTS COMMON-C-CONFIGURATION

IMPORTS K-CONTEXTS

END MODULE

MODULE COMMON-SEMANTIC-SYNTAX

IMPORTS COMMON-INCOMING-MODULES

SYNTAX $BaseValue ::= Nat$
 | Int
 | $Float$

SYNTAX $C ::= BaseValue$
 | $Type$
 | $Value$

SYNTAX $KResult ::= Value$
 | $Type$

DEFINE

true

$\frac{\forall_{Bool} \text{---}}{\text{true}}$

DEFINE

 —

$\frac{\forall_{Bool} \text{true}}{\text{true}}$

DEFINE

false \wedge_{Bool} —
 $\frac{\text{---}}{\text{false}}$

DEFINE

$$\frac{\text{— } \wedge_{Bool} \text{ false}}{\text{false}}$$

SYNTAX $K ::= \text{fromUnion}(Id)$

SYNTAX $Type ::= \text{typedDecl}(Type, Id)$

SYNTAX $K ::= \text{DeclType}(K, K)$ [strict(1)]

SYNTAX $Set ::= \text{setOfTypes}$

SYNTAX $K ::= \text{usualArithmeticConversion}(Type, Type)$
| $\text{callMain-aux}(K, Nat, Id, K)$ [strict(1)]
| $\text{initFunction}(K, K)$ [strict]
| $\text{populateFromGlobal}$
| $\text{checkValidLoc}(K)$
| $\text{checkDerefLoc}(K)$

SYNTAX $ListItem ::= \text{ListItem}(Bag)$

SYNTAX $K ::= \text{ListToK}(List)$ [function]
| $\text{Map}(Map)$

SYNTAX $Nat ::= \text{piece}(Nat, Nat)$

DEFINE

$$\frac{\text{isInt}(\text{piece}(\text{—}, \text{—}))}{\text{true}}$$

SYNTAX $Nat ::= \text{unknown}(Nat)$

DEFINE

$$\frac{\text{isInt}(\text{unknown}(\text{—}))}{\text{true}}$$

SYNTAX $KResult ::= \text{skipval}$

SYNTAX $K ::= \text{debug}$
| $\text{debug-m}(K)$
| discard

SYNTAX $Id ::= \text{File-Scope}$
| unnamedBitFields

SYNTAX $Nat ::= \text{loc}(K, K, K)$

DEFINE
 $\frac{\text{isInt}(\text{loc}(_, _, _))}{\text{true}}$

SYNTAX $K ::= K + \text{bits } K$ [function]

DEFINE
 $\frac{\text{loc}(\text{Base}, \text{Offset}, \text{BitOffset}) + \text{bits } N}{\text{loc}(\text{Base}, \text{Offset}, \text{BitOffset} + \text{Int } N)}$

SYNTAX $Value ::= \text{enumItem}(Id, Value)$

SYNTAX $K ::= \text{resolveReferences}$

SYNTAX $String ::= \text{toString}(K)$

SYNTAX $Type ::= \text{maxType}(Type, Type)$

SYNTAX $Nat ::= \text{bitRange}(Nat, Nat, Nat)$

SYNTAX $K ::= \text{fillToBytes}(K)$ [strict]

SYNTAX $Nat ::= \text{floorLoc}(Nat)$
| $\text{ceilingLoc}(Nat)$

SYNTAX $K ::= \text{readFunction}(Nat)$

SYNTAX $Type ::= \text{innerType}(Type)$ [function]

SYNTAX $K ::= \text{extractBitsFromList}(K, Nat, Nat)$ [strict(1)]

SYNTAX $Id ::= \text{typedef}(Id)$
| $\text{unnamed}(Nat)$

SYNTAX $Nat ::= \text{NullPointerConstant}$

DEFINE

$\frac{\text{isInt}(\text{NullPointerConstant})}{\text{true}}$

SYNTAX $Nat ::= \text{NullPointer}$

DEFINE

$\frac{\text{isInt}(\text{NullPointer})}{\text{true}}$

SYNTAX $Value ::= \text{emptyValue}$

SYNTAX $K ::= \text{allocate}(Type, K)$

| $\text{zero}(K)$
| $\text{zeroBlock}(Nat)$
| $\text{value}(K)$
| $\text{sizeofLocation}(K)$

SYNTAX $Type ::= \text{type}(K)$

SYNTAX $K ::= \text{flush}(Nat)$

| $\text{allocateType}(Nat, Type)$
| $\text{allocateTypeIfAbsent}(Nat, Type)$
| $\text{giveType}(Id, Type)$
| $\text{addToEnv}(Id, Nat)$
| $\text{read}(K, K)$ [strict(2)]
| $\text{write}(K, K)$ [strict(2)]
| $\text{writeByte}(Nat, K)$

SYNTAX $Bool ::= \text{isTypeCompatible}(K, K)$ [function]
| $\text{isPromoted}(Type)$ [function]

SYNTAX $Nat ::= \text{inc}(Nat)$

DEFINE

$\frac{\text{isNat}(\text{threadId}(N) +_{Int} M)}{\text{true}}$

DEFINE

$$\frac{\text{isNat}(\text{allocatedDuration} +_{\text{Int}} M)}{\text{true}}$$

SYNTAX $\text{Nat} ::= \text{threadId}(\text{Nat})$
 | allocatedDuration

SYNTAX $K ::= \text{initialize}(\text{Id}, \text{Type}, K)$

SYNTAX $\text{BagItem} ::= \text{mlength}(\text{Nat})$
 | mconst

SYNTAX $K ::= \text{makeUnwritable}(\text{Nat})$
 | $\text{makeUnwritableSubObject}(K)$
 | $\text{makeUnwritableVar}(K)$

CONTEXT: $\text{makeUnwritableSubObject}(\frac{\square}{\text{peval}(\square)})$

SYNTAX $K ::= \text{listToK}(K)$
 | $\text{klistToK}(\text{List}\{K\})$
 | UnknownCabsLoc
 | $\text{assert}(\text{Bool}, \text{Nat})$

SYNTAX $\text{SimpleType} ::= \text{bool}$
 | void
 | bool
 | char
 | short-int
 | int
 | long-int
 | long-long-int
 | float
 | double
 | long-double
 | signed-char
 | unsigned-char
 | $\text{unsigned-short-int}$
 | unsigned-int

- | unsigned-long-int
- | unsigned-long-long-int
- | no-type

SYNTAX $Type ::= \tau(Set, SimpleType)$

SYNTAX $Bool ::= isBasicType(K)$

SYNTAX $SimpleType ::= enumType(Id)$

- | $arrayType(Type, Int)$
- | $incompleteArrayType(Type)$
- | $flexibleArrayType(Type)$
- | $bitfieldType(Type, Nat)$
- | $functionType(Type, List\{KResult\})$
- | $pointerType(Type)$
- | $structType(Id)$
- | $unionType(Id)$
- | $qualifiedType(Type, K)$

SYNTAX $Type ::= unqualifyType(K)$ [function]

- | $removeStorageSpecifiers(K)$ [function]

SYNTAX $SimpleType ::= prototype(Type)$

- | $typedefType(Id, Type)$
- | $variadic$

SYNTAX $KResult ::= dataList(List\{K\})$

SYNTAX $K ::= sizeofType(K)$ [strict]

- | $bitSizeofType(K)$ [strict]
- | $byteSizeofType(K)$ [strict]

SYNTAX $Nat ::= bitsToBytes(K)$

SYNTAX $K ::= \perp(KLabel)$

SYNTAX $Bool ::= Set$ contains K [function]

SYNTAX $Set ::= assignmentLabels$

MACRO

assignmentLabels = Set(l(_ := _), l(_ * = _), l(_ / = _), l(_ % = _), l(_ + = _), l(_ - = _), l(_ >> = _), l(_ << = _), l(_ & = _), l(_ ^ = _), l(_ | = _))

SYNTAX $Set ::= \text{getModifiers}(K)$

SYNTAX $K ::= \text{AllowWrite}(K) \text{ [strict]}$

RULE

$$\frac{\text{AllowWrite}(\text{lv}(N, T))}{\text{lv}(N, \text{stripConst}(T))}$$

[anywhere]

SYNTAX $Type ::= \text{stripConst}(Type) \text{ [function]}$

DEFINE

$$\text{stripConst}(\text{t}(\text{Const } _ , _))$$

•

DEFINE

$$\frac{\text{stripConst}(\text{t}(S, T))}{\text{t}(S, T)}$$

when $\neg_{Bool}(\text{Const in } S)$

SYNTAX $K ::= \text{bind}(\text{List}\{K\text{Result}\}, \text{List}\{K\text{Result}\})$

SYNTAX $Value ::= \text{List}\{K\} : Type$
| $\text{lv}(\text{List}\{K\}, Type)$

SYNTAX $K ::= \text{concretize}(Type, K) \text{ [strict(2)]}$

SYNTAX $Value ::= \text{functionObject}(Id, Type, K)$
| $\text{functionPrototype}(Id, Type)$

SYNTAX $Char ::= \text{firstChar}(String)$
| $\text{nthChar}(String, Nat)$

SYNTAX $String ::= \text{butFirstChar}(String)$

SYNTAX $Nat ::= \text{charToAscii}(String)$

SYNTAX $Char ::= \text{stringToChar}(String)$

SYNTAX $Nat ::= \text{asciiCharString}(String)$

SYNTAX $List\{K\} ::= Nat \text{ to } Nat$ [function]

SYNTAX $K ::= \text{cast}(K, K)$ [function strict]

CONTEXT: $\text{cast}(-, \frac{\square}{\text{reval}(\square)})$

SYNTAX $K ::= \text{arithInterpret}(Type, BaseValue)$ [function]
 | $\text{interpret}(Type, K)$ [function]

SYNTAX $Set ::= \text{unsignedIntegerTypes}$
 | $\text{signedIntegerTypes}$

SYNTAX $Bool ::= \text{hasIntegerType}(Type)$ [function]
 | $\text{isFloatType}(Type)$ [function]
 | $\text{hasUnsignedIntegerType}(Type)$ [function]
 | $\text{hasSignedIntegerType}(Type)$ [function]

SYNTAX $K ::= \text{typeof}(K)$
 | $\text{writeToFD}(Nat, Nat)$
 | $\text{writeToFD}(Nat, String)$
 | $\text{readFromFD}(Nat)$
 | $\text{readFromFD}(Nat, Nat)$
 | $\text{calculateGotoMap}(Id, K)$

SYNTAX $Bool ::= \text{isCharType}(Type)$ [function]
 | $\text{isWCharType}(Type)$ [function]
 | $\text{isPointerType}(Type)$ [function]
 | $\text{isArrayType}(Type)$ [function]
 | $\text{isBoolType}(Type)$ [function]
 | $\text{isStructType}(Type)$ [function]
 | $\text{isUnionType}(Type)$ [function]
 | $\text{isAggregateType}(Type)$ [function]
 | $\text{isFunctionType}(Type)$ [function]

- | isFunctionPointerType(*Type*) [function]
- | isBitfieldType(*Type*) [function]
- | isExternType(*Type*) [function]
- | isStaticType(*Type*) [function]
- | isConstType(*Type*) [function]
- | isIncompleteType(*Type*) [function]
- | isArithmeticType(*Type*) [function]

SYNTAX *K* ::= aggregateInfo(*List*{*K*}, *Map*, *Map*)

SYNTAX *Nat* ::= getFieldOffset(*Id*, *K*) [function]

SYNTAX *Type* ::= getFieldtype(*Id*, *K*) [function]

SYNTAX *Bool* ::= isArithBinConversionOp(*KLabel*) [function]
 | isArithUnaryOp(*KLabel*) [function]

SYNTAX *K* ::= kpair(*K*, *K*)

SYNTAX *Type* ::= promote(*K*) [function]

SYNTAX *K* ::= argPromote(*K*) [function]
 | extractField(*List*{*K*}, *K*, *Id*)
 | allocString(*Nat*, *String*)
 | allocWString(*Nat*, *List*{*K*})
 | sequencePoint
 | handleBuiltin(*Id*, *Type*)

SYNTAX *Int* ::= min(*Type*) [function]
 | max(*Type*) [function]

SYNTAX *K* ::= alloc(*K*, *K*)
 | realloc(*K*, *K*, *K*, *K*)

SYNTAX *KResult* ::= initValue(*Id*, *Type*, *K*)

SYNTAX *K* ::= figureInit(*Id*, *K*, *K*) [strict(2)]
 | append(*Nat*, *Nat*, *Value*)
 | deleteBlock(*Nat*)

| deleteSizedBlock(*Nat*, *Nat*)

SYNTAX *Bool* ::= isConcreteNumber(*Int*) [function]

END MODULE

MODULE COMMON-C-SETTINGS

IMPORTS COMMON-SEMANTIC-SYNTAX

RULE

$$\frac{\text{char}}{\text{signed-char}} \\ \text{[anywhere]}$$

SYNTAX #*NzNat* ::= numBitsPerByte [function]

SYNTAX *Nat* ::= numBytes(*Type*) [function]
| numBits(*Type*) [function]

DEFINE NUMBITSPERBYTE

$$\frac{\text{numBitsPerByte}}{8}$$

DEFINE NUMBYTES-BOOL

$$\frac{\text{numBytes}(t(-, \text{bool}))}{1}$$

DEFINE NUMBYTES-SIGNED-CHAR

$$\frac{\text{numBytes}(t(-, \text{signed-char}))}{1}$$

DEFINE NUMBYTES-SHORT-INT

$$\frac{\text{numBytes}(t(-, \text{short-int}))}{2}$$

DEFINE NUMBYTES-INT

$$\frac{\text{numBytes}(t(-, \text{int}))}{4}$$

DEFINE **NUMBYTES-LONG-INT**

$$\frac{\text{numBytes}(t(_, \text{long-int}))}{4}$$

DEFINE **NUMBYTES-LONG-LONG-INT**

$$\frac{\text{numBytes}(t(_, \text{long-long-int}))}{8}$$

DEFINE **NUMBYTES-FLOAT**

$$\frac{\text{numBytes}(t(_, \text{float}))}{4}$$

DEFINE **NUMBYTES-DOUBLE**

$$\frac{\text{numBytes}(t(_, \text{double}))}{8}$$

DEFINE **NUMBYTES-LONG-DOUBLE**

$$\frac{\text{numBytes}(t(_, \text{long-double}))}{16}$$

DEFINE **NUMBYTES-ENUM**

$$\frac{\text{numBytes}(t(S, \text{enumType}(X)))}{\text{numBytes}(t(S, \text{int}))}$$

SYNTAX *Int* ::= *cfg:mtxPlain* [function]

DEFINE **CFG-MTXPLAIN**

$$\frac{\text{cfg:mtxPlain}}{0}$$

SYNTAX *Type* ::= *cfg:sizeut* [function]

DEFINE **CFG-SIZE-T**

$$\frac{\text{cfg:sizeut}}{t(\bullet, \text{unsigned-int})}$$

SYNTAX *Type* ::= *cfg:wcharut* [function]

DEFINE **CFG-WCHAR-T**

$$\frac{\text{cfg:wcharut}}{t(\bullet, \text{int})}$$

SYNTAX *SimpleType* ::= *simpleType*(*Type*) [function]

```

DEFINE
  
$$\frac{\text{simpleType}(t(-, T))}{T}$$


SYNTAX Type ::= cfg:largestUnsigned [function]

DEFINE CFG-LARGESTUNSIGNED
  
$$\frac{\text{cfg:largestUnsigned}}{t(\bullet, \text{unsigned-long-long-int})}$$


SYNTAX Nat ::= cfg:ptrsize [function]

DEFINE CFG-PTRSIZE
  
$$\frac{\text{cfg:ptrsize}}{4}$$


SYNTAX Type ::= cfg:ptrdiffut [function]

DEFINE CFG-PTRDIFF-T
  
$$\frac{\text{cfg:ptrdiffut}}{t(\bullet, \text{int})}$$


DEFINE MIN
  
$$\frac{\min(t(S, \text{enumType}(-)))}{\min(t(S, \text{int}))}$$


DEFINE MAX
  
$$\frac{\max(t(S, \text{enumType}(-)))}{\max(t(S, \text{int}))}$$


SYNTAX Int ::= rank(Type) [function]

END MODULE

MODULE COMMON-NOHELPER-INCLUDE

  IMPORTS COMMON-SEMANTIC-SYNTAX

  IMPORTS COMMON-C-SETTINGS

END MODULE

```

MODULE COMMON-INCLUDE

IMPORTS COMMON-NOHELPER-INCLUDE

IMPORTS COMMON-C-HELPERS

END MODULE

MODULE COMMON-C-SEMANTICS-MISC

IMPORTS COMMON-INCLUDE

DEFINE

$$\frac{\text{loc}(Base, BiOff) +_{Int} Offset}{\text{loc}(Base, BiOff +_{Int} Offset, BiOff)}$$

DEFINE

$$\frac{(\text{threadId}(N) +_{Int} M) +_{Int} N'}{\text{threadId}(N) +_{Int} (M +_{Int} N')}$$

DEFINE

$$\frac{(\text{allocatedDuration} +_{Int} M) +_{Int} N'}{\text{allocatedDuration} +_{Int} (M +_{Int} N')}$$

DEFINE

$$\frac{\text{inc}(\text{loc}(N, M, M'))}{\text{loc}(N +_{Int} 1, M, M')}$$

RULE UNKNOWN-LOC

$$\frac{\text{CabsLoc}(\text{"cabs loc unknown"}, -_{Int} 10, -_{Int} 10, 0)}{\text{UnknownCabsLoc}}$$

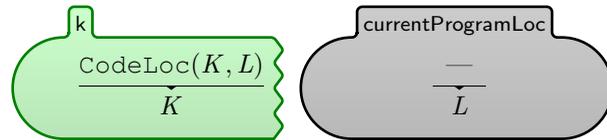
[anywhere]

RULE EXPRESSION-LOC

$$\frac{\text{ExpressionLoc}(K, -)}{K}$$

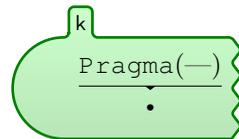
[anywhere]

RULE CODELOC-K



(n1570) §6.10.6 ¶1 A preprocessing directive of the form `#pragma pp-tokensoptnew-line` where the preprocessing token STDC does not immediately follow pragma in the directive (prior to any macro replacement) causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such pragma that is not recognized by the implementation is ignored.

RULE PRAGMA



RULE

$\frac{\text{AttributeWrapper}(K, -)}{K}$
[anywhere]

DEFINE

$\frac{\text{loc}(\text{Base}, \text{Offset}, \text{BitOffset})}{\text{loc}(\text{Base}, \text{Offset} +_{Int} (\text{BitOffset} \div_{Int} \text{numBitsPerByte}), \text{BitOffset} \%_{Int} \text{numBitsPerByte})}$
when $\text{BitOffset} \geq_{Int} \text{numBitsPerByte}$

RULE

$\frac{\text{Identifier}(\text{"_missing_field_name"})}{\#NoName}$
[anywhere]

END MODULE

MODULE COMMON-C-SEMANTICS

IMPORTS COMMON-INCLUDE

IMPORTS COMMON-C-SEMANTICS-MISC

IMPORTS COMMON-C-EXPRESSIONS

IMPORTS COMMON-C-STATEMENTS

IMPORTS COMMON-C-DECLARATIONS

IMPORTS COMMON-C-TYPING

SYNTAX $Bag ::= eval(K)$
 | $eval(K, List\{K\}, String, Int)$

SYNTAX $K ::= callMain(Nat, K)$
 | $incomingArguments(List\{K\})$

SYNTAX $KLabel ::= TranslationUnitName(String)$

END MODULE

MODULE COMMON-SEMANTICS-HELPERS-INCLUDE

IMPORTS COMMON-NOHELPER-INCLUDE

SYNTAX $Nat ::= Nat\ bit :: Nat$

END MODULE

MODULE COMMON-SEMANTICS-HELPERS-MISC

IMPORTS COMMON-SEMANTICS-HELPERS-INCLUDE

SYNTAX $K ::= firstLoc(K)$ [function]

DEFINE

$firstLoc(ThreadId)$

 $loc(threadId(ThreadId) +_{Int} 0, 0, 0)$

SYNTAX $Nat ::= base(Nat)$ [function]

DEFINE

$$\frac{\text{base}(\text{loc}(\text{Base}, -, -))}{\text{Base}}$$

SYNTAX $\text{Set} ::= \text{gatherInnerCells}(\text{Bag}, \text{CellLabel})$ [function]

DEFINE

$$\text{gatherInnerCells}(\frac{\langle L' \rangle \langle L \rangle K \langle / L \rangle - \langle / L' \rangle - , L}{\bullet}, \frac{\bullet}{K})$$

DEFINE

$$\frac{\text{gatherInnerCells}(\bullet, -)}{\bullet}$$

SYNTAX $\text{List} ::= \text{stringToList}(\text{String})$ [function]

SYNTAX $\text{String} ::= \text{listToString}(\text{List})$ [function]

DEFINE

$$\frac{\text{stringToList}("")}{\bullet}$$

DEFINE

$$\frac{\text{stringToList}(S)}{\text{firstChar}(S) \text{ stringToList}(\text{butFirstChar}(S)) \text{ when } S \neq_{\text{String}} ""}$$

DEFINE

$$\frac{\text{listToString}(\bullet)}{""}$$

DEFINE

$$\frac{\text{listToString}(S \ L)}{S +_{\text{String}} \text{listToString}(L)}$$

SYNTAX $\text{Bool} ::= \text{isUnknown}(K)$ [function]

DEFINE **ISUNKNOWN-PIECE**

$$\frac{\text{isUnknown}(\text{piece}(-, -))}{\text{true}}$$

DEFINE ISUNKNOWN-PTR

$$\frac{\text{isUnknown}(\text{loc}(-, -, -))}{\text{false}}$$

DEFINE ISUNKNOWN-INT

$$\frac{\text{isUnknown}(I)}{\text{false}}$$

when $(I \leq_{Int} 0)$
 $\vee_{Bool} (I >_{Int} 0)$

DEFINE

$$\frac{\text{loc}(N, M, 0) <_{Int} \text{loc}(N, M', 0)}{\text{true}}$$

 when $M <_{Int} M'$

DEFINE

$$\frac{\text{loc}(N, M, 0) \leq_{Int} \text{loc}(N, M', 0)}{\text{true}}$$

 when $M \leq_{Int} M'$

DEFINE

$$\frac{\text{loc}(N, M, 0) >_{Int} \text{loc}(N, M', 0)}{\text{true}}$$

 when $M >_{Int} M'$

DEFINE

$$\frac{\text{loc}(N, M, 0) \geq_{Int} \text{loc}(N, M', 0)}{\text{true}}$$

 when $M \geq_{Int} M'$

SYNTAX $K ::= \text{simplifyTruth}(K)$ [function]

DEFINE

$$\frac{\text{simplifyTruth}(K)}{K \neq 0 : t(\bullet, \text{int})}$$

SYNTAX $Bool ::= \text{isNotTruthValue}(Value)$ [function]

```

DEFINE
  
$$\frac{\text{isNotTruthValue}(V : t(-, T))}{\frac{(T \neq_K \text{int})}{\forall_{Bool} ((V \neq_K 0) \wedge_{Bool} (V \neq_K 1))}}$$

SYNTAX  $K ::= \text{getIdOfDeclaration}(K)$  [function]
      |  $\text{getIdOfName}(K)$  [function]
DEFINE
  
$$\frac{\text{getIdOfDeclaration}(\text{DeclarationDefinition}(\text{InitNameGroup}(-, \text{List}(K))))}{\text{getIdOfName}(K)}$$

DEFINE
  
$$\frac{\text{getIdOfName}(\text{InitName}(K, -))}{\text{getIdOfName}(K)}$$

DEFINE
  
$$\frac{\text{getIdOfName}(\text{SingleName}(-, K))}{\text{getIdOfName}(K)}$$

DEFINE
  
$$\frac{\text{getIdOfName}(\text{Name}(X, -))}{X}$$

SYNTAX  $K ::= \text{fillToBytes-aux}(K, \text{List}\{K\})$  [function]
DEFINE FILLTOBYTES-START
  
$$\frac{\text{fillToBytes}(\text{dataList}(L))}{\text{fillToBytes-aux}(\text{dataList}(L), \bullet)}$$

DEFINE FILLTOBYTES-FOUNDBYTE
  
$$\frac{\text{fillToBytes-aux}(\text{dataList}(L \text{ , piece}(N, Len)), L')}{\text{fillToBytes-aux}(\text{dataList}(L), \text{piece}(N, Len) \text{ , } L')}$$

  when  $Len ==_{Int} \text{numBitsPerByte}$ 
DEFINE FILLTOBYTES-ADDBIT
  
$$\frac{\text{fillToBytes-aux}(\text{dataList}(\text{piece}(N, Len)), L')}{\text{fillToBytes-aux}(\text{dataList}(\text{piece}(0, 1) \text{ , piece}(N, Len)), L')}$$

  when  $Len <_{Int} \text{numBitsPerByte}$ 

```

DEFINE **FILLTOBYTES-COMBINEBITS**

$$\frac{\text{fillToBytes-aux}(\text{dataList}(L, \text{piece}(N, Len), \text{piece}(N', Len')), L')}{\text{fillToBytes-aux}(\text{dataList}(L, \text{piece}(\text{piece}(N, Len) \text{ bit} :: \text{piece}(N', Len'), Len +_{Int} Len')), L')}$$

when $(Len +_{Int} Len') \leq_{Int} \text{numBitsPerByte}$

DEFINE **FILLTOBYTES-DONE**

$$\frac{\text{fillToBytes-aux}(\text{dataList}(\bullet), L)}{\text{dataList}(L)}$$

DEFINE

$$\frac{\text{piece}(\text{bitRange}(N, sNatTo, To'), Len) \text{ bit} :: \text{piece}(\text{bitRange}(N, From, To), Len')}{\text{piece}(\text{bitRange}(N, From, To'), Len +_{Int} Len')}$$

when $((Len +_{Int} Len') \leq_{Int} \text{numBitsPerByte}) \wedge_{Bool} (sNatTo ==_{Int} (To +_{Int} 1))$

DEFINE

$$\frac{\text{piece}(N \text{ bit} :: N', Len) \text{ bit} :: \text{piece}(N'', Len')}{\text{piece}((N \text{ bit} :: N') \text{ bit} :: \text{piece}(N'', Len'), Len +_{Int} Len')}$$

DEFINE

$$\frac{\text{piece}(N'', Len') \text{ bit} :: \text{piece}(N \text{ bit} :: N', Len)}{\text{piece}((\text{piece}(N'', Len') \text{ bit} :: N) \text{ bit} :: N', Len +_{Int} Len')}$$

DEFINE

$$\frac{\text{bitRange}(N \text{ bit} :: \text{piece}(\text{---}, Len), Pos, Pos)}{\text{bitRange}(N, \text{absInt}(Pos -_{Int} Len), \text{absInt}(Pos -_{Int} Len))}$$

when $(Pos >_{Int} 0) \wedge_{Bool} ((Pos -_{Int} Len) \geq_{Int} 0)$

DEFINE

$$\frac{\text{bitRange}(\text{--- bit} :: \text{piece}(N, 1), 0, 0)}{\text{piece}(N, 1)}$$

DEFINE

$$\frac{\text{bitRange}(\text{piece}(N, 1), 0, 0)}{\text{piece}(N, 1)}$$

DEFINE

$$\frac{\text{bitRange}(\text{piece}(\text{bitRange}(N, Start, End), Len), 0, 0)}{\text{bitRange}(\text{piece}(\text{bitRange}(N, Start, Start), 1), 0, 0)}$$

when $(Start +_{Int} Len) ==_{Int} (End +_{Int} 1)$

DEFINE

$$\frac{\text{bitRange}(N, Pos, Pos)}{1 \&_{Int} (N \gg_{Int} Pos)}$$

when $N \geq_{Int} 0$

DEFINE

$$\frac{\text{bitRange}(\text{piece}(N, 1), Pos, Pos)}{1 \&_{Int} (N \gg_{Int} Pos)}$$

when $N \geq_{Int} 0$

DEFINE

$$\frac{\text{bitRange}(N, 0, To)}{N}$$

when $(To +_{Int} 1) ==_{Int} \text{numBitsPerByte}$

DEFINE

$$\frac{\text{bitRange}(- \text{bit} :: \text{piece}(N, Len), Start, End)}{\text{bitRange}(\text{piece}(N, Len), Start, End)}$$

when $(End +_{Int} 1) \leq_{Int} Len$

DEFINE

$$\frac{\text{bitRange}(\text{piece}(N, sNatEnd), 0, End)}{\text{piece}(N, End +_{Int} 1)}$$

when $sNatEnd ==_{Int} (End +_{Int} 1)$

DEFINE

$$\frac{\text{bitRange}(- \text{bit} :: \text{piece}(N, sNatEnd), 0, End)}{\text{piece}(N, End +_{Int} 1)}$$

when $sNatEnd ==_{Int} (End +_{Int} 1)$

DEFINE

$$\frac{\text{bitRange}(\text{piece}(N, Len), Pos, Pos)}{(N \gg_{Int} Pos) \&_{Int} 1}$$

when $N \geq_{Int} 0$

SYNTAX $K ::= \text{extractField-pre}(List\{K\}, Type, Nat, K) [\text{strict}(4)]$
 $| \text{extractField-aux}(List\{K\}, Type, Nat, Nat, List\{K\})$

RULE **EXTRACTFIELD-START**

$$\frac{\text{extractField}(L, t(-, L(S)), F)}{\text{extractField-pre}(L, T, \text{Offset}, \text{bitSizeOfType}(T))}$$

when $(L ==_{KLabel} \text{unionType})$
 $\vee_{Bool} (L ==_{KLabel} \text{structType})$

structs
 $S \mapsto \text{aggregateInfo}(-, - (F \mapsto T), - (F \mapsto \text{Offset}))$

RULE

$$\frac{\text{extractField-pre}(L, T, \text{Offset}, \text{Len} : -)}{\text{concretize}(T, \text{fillToBytes}(\text{extractBitsFromList}(\text{dataList}(L), \text{Offset}, \text{Len})))}$$

DEFINE

$$\frac{\text{isConcreteNumber}(\text{loc}(-, -, -))}{\text{false}}$$

DEFINE

$$\frac{\text{isConcreteNumber}(I)}{\text{true}}$$

when $(I \leq_{Int} 0)$
 $\vee_{Bool} (I >_{Int} 0)$

RULE **DISCARD**

$$\frac{V \curvearrowright \text{discard}}{\bullet}$$

DEFINE

$$\frac{\text{bitsToBytes}(N)}{\text{absInt}(N \div_{Int} \text{numBitsPerByte})}$$

when numBitsPerByte dividesInt N

DEFINE

$$\frac{\text{bitsToBytes}(N)}{\text{absInt}((N \div_{Int} \text{numBitsPerByte}) +_{Int} 1)}$$

when $\neg_{Bool} (\text{numBitsPerByte}$ dividesInt $N)$

```

DEFINE
  numBytes(t(—, unsigned-char))
  numBytes(t(•, signed-char))
DEFINE
  numBytes(t(—, unsigned-short-int))
  numBytes(t(•, short-int))
DEFINE
  numBytes(t(—, unsigned-int))
  numBytes(t(•, int))
DEFINE
  numBytes(t(—, unsigned-long-int))
  numBytes(t(•, long-int))
DEFINE
  numBytes(t(—, unsigned-long-long-int))
  numBytes(t(•, long-long-int))
DEFINE
  numBits(t(S, T))
  numBytes(t(S, T)) *Int numBitsPerByte
  when (getKLabel(T)) =/= KLabel bitfieldType
DEFINE
  numBits(t(—, bitfieldType(—, N)))
  N
DEFINE
  min(t(—, bool))
  0
DEFINE
  max(t(—, bool))
  1
DEFINE
  min(t(—, signed-char))
  0 -Int (2 ^Int (absInt (numBits(t(•, signed-char)) -Int 1)))
DEFINE
  max(t(—, signed-char))
  (2 ^Int (absInt (numBits(t(•, signed-char)) -Int 1))) -Int 1

```

```

DEFINE
  
$$\frac{\min(t(-, \text{short-int}))}{0 -_{Int} (2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{short-int})) -_{Int} 1)))}$$

DEFINE
  
$$\frac{\max(t(-, \text{short-int}))}{(2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{short-int})) -_{Int} 1))) -_{Int} 1}$$

DEFINE
  
$$\frac{\min(t(-, \text{int}))}{0 -_{Int} (2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{int})) -_{Int} 1)))}$$

DEFINE
  
$$\frac{\max(t(-, \text{int}))}{(2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{int})) -_{Int} 1))) -_{Int} 1}$$

DEFINE
  
$$\frac{\min(t(-, \text{long-int}))}{0 -_{Int} (2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{long-int})) -_{Int} 1)))}$$

DEFINE
  
$$\frac{\max(t(-, \text{long-int}))}{(2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{long-int})) -_{Int} 1))) -_{Int} 1}$$

DEFINE
  
$$\frac{\min(t(-, \text{long-long-int}))}{0 -_{Int} (2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{long-long-int})) -_{Int} 1)))}$$

DEFINE
  
$$\frac{\max(t(-, \text{long-long-int}))}{(2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{long-long-int})) -_{Int} 1))) -_{Int} 1}$$

DEFINE
  
$$\frac{\min(t(-, \text{unsigned-char}))}{0}$$

DEFINE
  
$$\frac{\max(t(-, \text{unsigned-char}))}{(2^{Int} (\text{absInt} (\text{numBits}(t(\bullet, \text{unsigned-char}))) -_{Int} 1)}$$

DEFINE
  
$$\frac{\min(t(-, \text{unsigned-short-int}))}{0}$$


```

```

DEFINE
  
$$\frac{\max(t(-, \text{unsigned-short-int}))}{(2^{\text{Int}} (\text{absInt numBits}(t(\bullet, \text{unsigned-short-int})))) - \text{Int } 1}$$

DEFINE
  
$$\frac{\min(t(-, \text{unsigned-int}))}{0}$$

DEFINE
  
$$\frac{\max(t(-, \text{unsigned-int}))}{(2^{\text{Int}} (\text{absInt numBits}(t(\bullet, \text{unsigned-int})))) - \text{Int } 1}$$

DEFINE
  
$$\frac{\min(t(-, \text{unsigned-long-int}))}{0}$$

DEFINE
  
$$\frac{\max(t(-, \text{unsigned-long-int}))}{(2^{\text{Int}} (\text{absInt numBits}(t(\bullet, \text{unsigned-long-int})))) - \text{Int } 1}$$

DEFINE
  
$$\frac{\min(t(-, \text{unsigned-long-long-int}))}{0}$$

DEFINE
  
$$\frac{\max(t(-, \text{unsigned-long-long-int}))}{(2^{\text{Int}} (\text{absInt numBits}(t(\bullet, \text{unsigned-long-long-int})))) - \text{Int } 1}$$

DEFINE
  
$$\frac{\text{stringToChar}(C)}{C}$$

DEFINE
  
$$\frac{\text{asciiCharString}(S)}{\text{asciiString}(\text{stringToChar}(S))}$$

DEFINE FIRSTCHAR
  
$$\frac{\text{firstChar}(S)}{\text{substrString}(S, 0, 1)}$$

DEFINE NTHCHAR
  
$$\frac{\text{nthChar}(S, N)}{\text{substrString}(S, N, 1)}$$


```

```

DEFINE CHARToASCII
  
$$\frac{\text{charToAscii}(C)}{\text{asciiString}(C)}$$

DEFINE BUTFIRSTCHAR
  
$$\frac{\text{butFirstChar}(S)}{\text{substrString}(S, 1, \text{lengthString}(S))}$$

SYNTAX String ::= toUpperCase(String) [function]
SYNTAX Char ::= toUpperCase(Char) [function]
DEFINE
  
$$\frac{\text{toUpperCase}(S)}{\text{toUpperCase}(\text{firstChar}(S)) +_{\text{String}} \text{toUpperCase}(\text{butFirstChar}(S))}$$

  when  $S \neq_{\text{String}} ""$ 
DEFINE
  
$$\frac{\text{toUpperCase}("")}{""}$$

DEFINE
  
$$\frac{\text{toUpperCase}(C)}{C}$$

  when  $\bigvee_{\text{Bool}} (\text{asciiString}(C) <_{\text{Int}} \text{asciiString}("a"))$ 
   $\bigvee_{\text{Bool}} (\text{asciiString}(C) >_{\text{Int}} \text{asciiString}("z"))$ 
DEFINE
  
$$\frac{\text{toUpperCase}(C)}{\text{charString}(\text{absInt}(\text{asciiString}(C) -_{\text{Int}} (\text{asciiString}("a") -_{\text{Int}} \text{asciiString}("A"))))}$$

  when  $(\text{asciiString}(C) \geq_{\text{Int}} \text{asciiString}("a")) \wedge_{\text{Bool}} (\text{asciiString}(C) \leq_{\text{Int}} \text{asciiString}("z"))$ 
SYNTAX K ::= getString(K)
  | getString-aux(K, String) [strict(1)]
RULE GETSTRING-START
  
$$\frac{\text{getString}(K)}{\text{getString-aux}(K, "")}$$

  [anywhere]

```

SYNTAX $Value ::= \text{str}(String)$

RULE GETSTRING-PRE

$$\frac{k \quad \cdot \quad \curvearrowright \text{getString-aux}(Loc : -, S)}{\text{read}(Loc, t(\cdot, \text{char}))} \quad \frac{}{Loc +_{Int} 1}$$

RULE GETSTRING

$$\frac{k \quad N : - \curvearrowright \text{getString-aux}(Loc : -, S)}{\text{getString-aux}(Loc : t(\cdot, \text{pointerType}(t(\cdot, \text{unsigned-char}))), S +_{String} \text{charString}(N))}$$

when $N \neq_{Int} 0$

RULE GETSTRING-DONE

$$\frac{k \quad 0 : - \curvearrowright \text{getString-aux}(Loc : -, S)}{\text{str}(S)}$$

SYNTAX $K ::= \text{writeString}(K, String) [\text{strict}(1)]$
| $\text{writeWString}(K, List\{K\}) [\text{strict}(1)]$

RULE WRITE-STRING

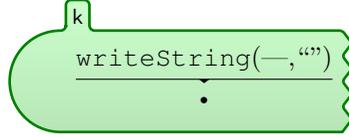
$$\frac{k \quad \text{writeString}(Loc : T, S)}{(* Loc : t(\cdot, \text{pointerType}(t(\cdot, \text{char})))) := \text{charToAscii}(\text{firstChar}(S)) : t(\cdot, \text{char}); \curvearrowright \text{writeString}(Loc +_{Int} 1 : T, \text{butFirstChar}(S))}$$

when $S \neq_{String} ""$

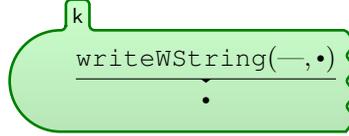
RULE WRITE-WSTRING

$$\frac{k \quad \text{writeWString}(Loc : T, N, S)}{(* Loc : t(\cdot, \text{pointerType}(\text{cfg:wcharut})) := N : \text{cfg:wcharut}; \curvearrowright \text{writeWString}(Loc +_{Int} 4 : T, S)}$$

RULE **WRITE-EMPTY-STRING**



RULE **WRITE-EMPTY-WSTRING**



SYNTAX $String ::= \text{pointerToString}(Nat) \text{ [function]}$

DEFINE **POINTERTOSTRING**

$\text{pointerToString}(\text{loc}(N, M, 0))$

$\frac{}{((("[\text{sym}('' +_{String} \text{subPointerToString}(N)) +_{String} ' ') +_{String} \text{Int2String}(M)) +_{String} ''])}$

SYNTAX $String ::= \text{subPointerToString}(Nat) \text{ [function]}$

DEFINE **SUBPOINTERTOSTRING-AUTO**

$\text{subPointerToString}(\text{threadId}(N) +_{Int} N')$

$\frac{}{("threadId('' +_{String} \text{Int2String}(N)) +_{String} ' ') +_{Int} N'}$

when $N \neq_K \text{allocatedDuration}$

DEFINE **SUB-POINTERTOSTRING-ALLOCATED**

$\text{subPointerToString}(\text{threadId}(\text{allocatedDuration}) +_{Int} N')$

$\frac{}{"threadId(\text{allocatedDuration}) +_{Int} " +_{String} \text{Int2String}(N')}$

DEFINE **POINTERTOSTRING-DONE**

$\text{pointerToString}(\text{NullPointer})$

$\frac{}{"NullPointer"}$

DEFINE

$N \text{ to } N$

$\frac{}{\bullet}$

DEFINE

$N \text{ to } N'$

$\frac{}{N \text{ , } ((N +_{Int} 1) \text{ to } N')}$

when $N <_{Int} N'$

```

DEFINE
  
$$\frac{S \text{ contains } K}{\text{true}}$$

DEFINE
  
$$\frac{S \text{ } K_1 \text{ contains } K_2}{S \text{ contains } K_2}$$

  when  $K_1 \neq_K K_2$ 
DEFINE
  
$$\frac{\bullet \text{ contains } K}{\text{false}}$$

DEFINE
  
$$\frac{\text{hasIntegerType}(T)}{\text{hasUnsignedIntegerType}(T) \vee_{Bool} \text{hasSignedIntegerType}(T)}$$

DEFINE
  
$$\frac{\text{hasUnsignedIntegerType}(t(-, T))}{\text{true}}$$

  when unsignedIntegerTypes contains  $T$ 
DEFINE
  
$$\frac{\text{hasUnsignedIntegerType}(t(S, T))}{\text{false}}$$

  when  $((\text{getKLabel}(T)) \neq_{KLabel} \text{bitfieldType}) \wedge_{Bool} \left( (\text{setOfTypes contains } l(\text{getKLabel}(T))) \vee_{Bool} \text{isFloatType}(t(S, T)) \right)$ 
DEFINE
  
$$\frac{\text{hasUnsignedIntegerType}(t(-, \text{bitfieldType}(T, -)))}{\text{true}}$$

  when  $\text{hasUnsignedIntegerType}(T) ==_{Bool} \text{true}$ 
DEFINE
  
$$\frac{\text{hasUnsignedIntegerType}(t(-, \text{bitfieldType}(T, -)))}{\text{false}}$$

  when  $\text{hasUnsignedIntegerType}(T) ==_{Bool} \text{false}$ 

```

```

DEFINE
  hasSignedIntegerType(t(—, T))
    true
    when signedIntegerTypes contains T
DEFINE
  hasSignedIntegerType(t(—, enumType(—)))
    true
DEFINE
  hasSignedIntegerType(t(S, T))
    false
    when ((getKLabel(T)) /=KLabel bitfieldType)  $\wedge_{Bool}$   $\left( \begin{array}{l} \text{setOfTypes contains l(getKLabel(T))} \\ \vee_{Bool} \text{isFloatType(t(S, T))} \end{array} \right)$ 
DEFINE
  hasSignedIntegerType(t(—, bitfieldType(T, —)))
    true
    when hasSignedIntegerType(T) ==Bool true
DEFINE
  hasSignedIntegerType(t(—, bitfieldType(T, —)))
    false
    when hasSignedIntegerType(T) ==Bool false
DEFINE
  min(t(—, bitfieldType(T, N)))
    0
    when hasUnsignedIntegerType(T)
DEFINE
  max(t(—, bitfieldType(T, N)))
     $(2^{\wedge_{Int} N}) -_{Int} 1$ 
    when hasUnsignedIntegerType(T)
DEFINE
  min(t(—, bitfieldType(T, N)))
     $0 -_{Int} (2^{\wedge_{Int} (N -_{Int} 1)})$ 
    when hasSignedIntegerType(T)

```

DEFINE

$$\frac{\max(\text{t}(-, \text{bitfieldType}(T, N)))}{(2 \hat{_{Int}} (\text{absInt } (N -_{Int} 1))) -_{Int} 1}$$
 when hasSignedIntegerType(T)

DEFINE

$$\frac{\text{NullPointerConstant}}{0}$$

DEFINE

$$\frac{\text{piece}(N, Len) \text{ bit} :: \text{piece}(N', Len')}{\text{piece}((N \ll_{Int} Len') |_{Int} N', Len +_{Int} Len')}$$
 when ($N \geq_{Int} 0$) \wedge_{Bool} ($N' \geq_{Int} 0$)

DEFINE

$$\frac{\text{piece}(0, 0) \text{ bit} :: N}{N}$$

DEFINE

$$\frac{\text{piece}(\text{piece}(N, Len), Len)}{\text{piece}(N, Len)}$$

DEFINE

$$\frac{\text{value}(V : -)}{V}$$

DEFINE

$$\frac{\text{type}(- : T)}{T}$$

SYNTAX $K ::= \text{bitSizeofList}(\text{List}\{K\text{Result}\})$ [function]
 | $\text{bitSizeofList-aux}(K, \text{Nat}, \text{List}\{K\text{Result}\})$ [function strict(1)]

DEFINE

$$\frac{\text{bitSizeofList}(L)}{\text{bitSizeofList-aux}(\bullet, 0, L)}$$

DEFINE

$$\text{bitSizeofList-aux}\left(\frac{\bullet}{\text{bitSizeofType}(T)}, -, \frac{T}{\bullet}, -\right)$$

DEFINE

$$\text{bitSizeofList-aux}\left(\frac{\text{Len}' : -}{\bullet}, \frac{\text{Len}}{\text{Len} +_{\text{Int}} \text{Len}'}, -\right)$$

DEFINE

$$\frac{\text{bitSizeofList-aux}(\bullet, \text{Len}, \bullet)}{\text{Len} : \text{cfg:largestUnsigned}}$$

SYNTAX $K ::= \text{maxBitSizeofList}(\text{List}\{K\text{Result}\})$
 | $\text{maxBitSizeofList-aux}(\text{List}\{K\text{Result}\}, \text{Nat})$

RULE

$$\frac{\text{maxBitSizeofList}(L)}{\text{maxBitSizeofList-aux}(L, 0)}$$

[anywhere]

RULE

$$\frac{\text{maxBitSizeofList-aux}(T, L, N)}{\text{bitSizeofType}(T) \curvearrowright \text{maxBitSizeofList-aux}(L, N)}$$

RULE

$$\frac{N' : - \curvearrowright \text{maxBitSizeofList-aux}(L, \frac{N}{\max\text{Int}(N, N')})}{\bullet}$$

RULE

$$\frac{\text{maxBitSizeofList-aux}(\bullet, N)}{N : \text{cfg:largestUnsigned}}$$

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{arrayType}(T, N)))}{\text{bitSizeofType}(T) * N : \text{cfg:largestUnsigned}}$$

[anywhere]

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{flexibleArrayType}(T)))}{0 : \text{cfg:largestUnsigned}}$$

[anywhere]

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{functionType}(-, -)))}{\text{numBitsPerByte} : \text{cfg:largestUnsigned}}$$

[anywhere]

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{pointerType}(-)))}{\text{cfg:ptrsize} * \text{Int} \text{ numBitsPerByte} : \text{cfg:largestUnsigned}}$$

[anywhere]

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{bitfieldType}(-, N)))}{N : \text{cfg:largestUnsigned}}$$

[anywhere]

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{qualifiedType}(T, -)))}{\text{bitSizeofType}(T)}$$

[anywhere]

RULE

$$\frac{\text{bitSizeofType}(T)}{\text{numBits}(T) : \text{cfg}:\text{largestUnsigned}}$$
 when isBasicType(T)
 [anywhere]

RULE

$$\frac{\text{bitSizeofType}(\text{typedDecl}(T, -))}{\text{bitSizeofType}(T)}$$
 [anywhere]

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{structType}(S)))}{\text{bitSizeofList}(L)}$$
 structs

$$S \mapsto \text{aggregateInfo}(L, -, -)$$

RULE

$$\frac{\text{bitSizeofType}(t(-, \text{unionType}(S)))}{\text{maxBitSizeofList}(L)}$$
 structs

$$S \mapsto \text{aggregateInfo}(L, -, -)$$

DEFINE

$$\frac{\text{getFieldOffset}(F, \text{aggregateInfo}(-, -, - (F \mapsto N)))}{N}$$

DEFINE

$$\frac{\text{getFieldType}(F, \text{aggregateInfo}(-, -, - (F \mapsto T)))}{T}$$

DEFINE

$$\frac{\text{toString}(\text{Identifier}(S))}{S}$$

DEFINE

$$\frac{\text{toString}(S)}{\hat{S}}$$

DEFINE

$$\frac{\text{toString}(Num)}{\text{Int2String}(Num)}$$

```

DEFINE
  listToK(K)
  klistToK(K)
DEFINE
  klistToK(K , L)
  K ~ klistToK(L)
DEFINE
  klistToK(•)
  •

```

END MODULE

MODULE COMMON-C-HELPERS

IMPORTS COMMON-SEMANTICS-HELPERS-INCLUDE

IMPORTS COMMON-C-BITSIZE

IMPORTS COMMON-SEMANTICS-HELPERS-MISC

END MODULE

MODULE DYNAMIC-SEMANTIC-SYNTAX

IMPORTS COMMON-INCLUDE

END MODULE

MODULE DYNAMIC-INCLUDE

IMPORTS DYNAMIC-SEMANTIC-SYNTAX

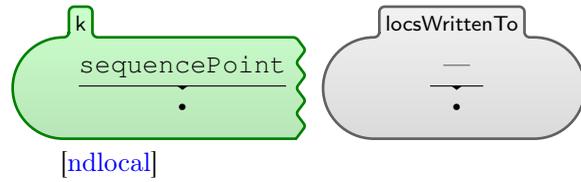
IMPORTS DYNAMIC-C-CONFIGURATION

END MODULE

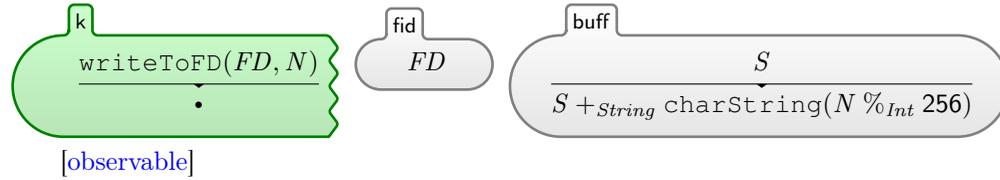
MODULE DYNAMIC-C-SEMANTICS-MISC

IMPORTS DYNAMIC-INCLUDE

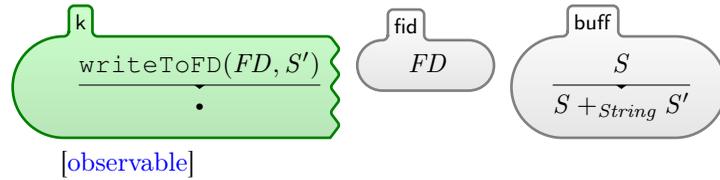
RULE SEQUENCEPOINT



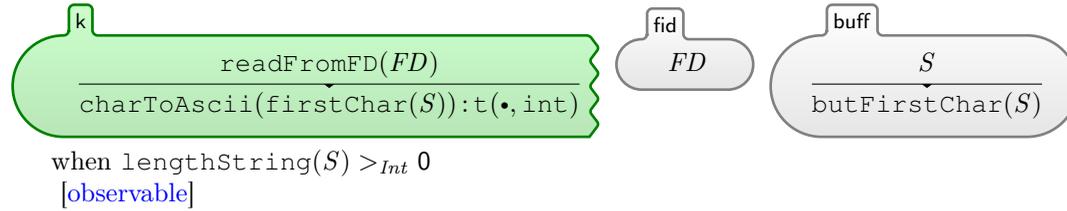
RULE WRITETOFD-CHAR



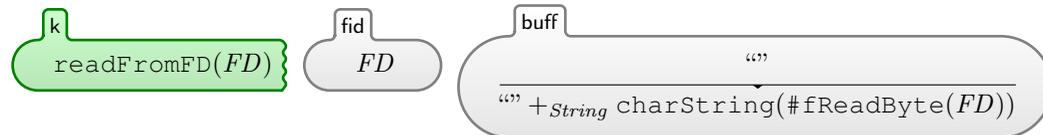
RULE WRITETOFD-STRING



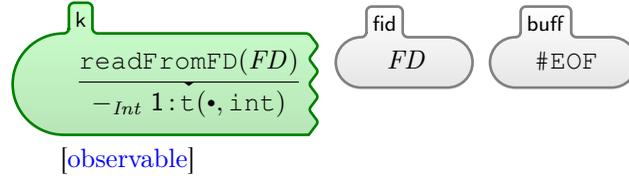
RULE READFROMFD-CHAR



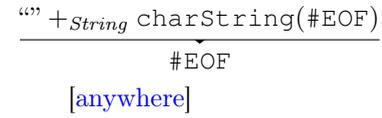
RULE READFROMFD-EMPTY-BUFF



RULE READFROMFD-EOF

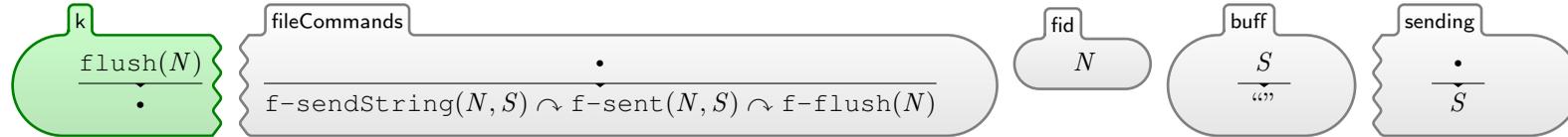


RULE MAKE-EOF

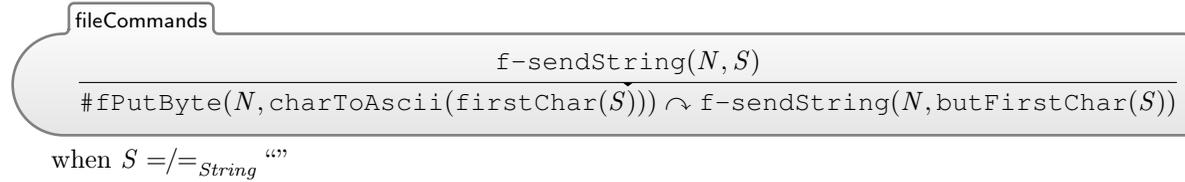


SYNTAX $K ::=$ f-sent($Nat, String$)
 | f-flush(Nat)
 | f-sendString($Nat, String$)

RULE FLUSH



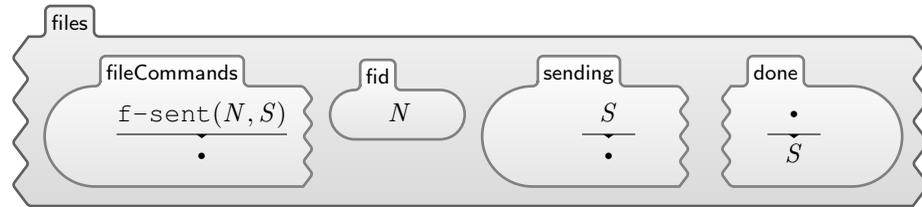
RULE SENDSTRING-ONE



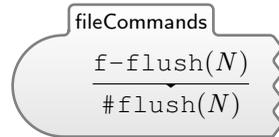
RULE SENDSTRING-DONE



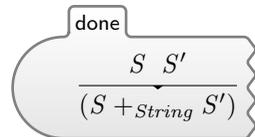
RULE F-SENT



RULE F-FLUSH



RULE COMBINE-DONE



SYNTAX $List\{K\} ::= \text{string2List}(String) [\text{function}]$
 $| \text{string2List-aux}(String, List\{K\}) [\text{function}]$

DEFINE

$\frac{\text{string2List}(S)}{\text{string2List-aux}(S, \bullet)}$

DEFINE

$\frac{\text{string2List-aux}(\text{"", } L)}{L}$

DEFINE

$\frac{\text{string2List-aux}(S, L)}{\text{string2List-aux}(\text{butFirstChar}(S), L, \text{firstChar}(S))}$
 when $S \neq_{String} \text{" "}$

END MODULE

MODULE DYNAMIC-C-SEMANTICS

IMPORTS COMMON-C-SEMANTICS

IMPORTS DYNAMIC-INCLUDE
IMPORTS DYNAMIC-C-SEMANTICS-MISC
IMPORTS DYNAMIC-C-EXPRESSIONS
IMPORTS DYNAMIC-C-ERRORS
IMPORTS DYNAMIC-C-TYPING
IMPORTS DYNAMIC-C-DECLARATIONS
IMPORTS DYNAMIC-C-MEMORY
IMPORTS DYNAMIC-C-STATEMENTS
IMPORTS DYNAMIC-C-CONVERSIONS
IMPORTS DYNAMIC-C-STANDARD-LIBRARY

(n1570) §5.1.2.2.1 ¶2 If they are declared, the parameters to the main function shall obey the following constraints:

- The value of `argc` shall be nonnegative.
- `argv[argc]` shall be a null pointer.
- If the value of `argc` is greater than zero, the array members `argv[0]` through `argv[argc-1]` inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.
- If the value of `argc` is greater than zero, the string pointed to by `argv[0]` represents the program name; `argv[0][0]` shall be the null character if the program name is not available from the host environment. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the program parameters.
- The parameters `argc` and `argv` and the strings pointed to by the `argv` array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

SYNTAX $K ::= \text{incomingArguments-aux}(List\{K\}, Nat)$

RULE

$$\frac{\text{incomingArguments}(L)}{\text{incomingArguments-aux}(L, 0)}$$

RULE

$$\frac{\text{incomingArguments-aux}(S, L, N)}{(\text{Identifier}(\text{"\#incomingArgumentsArray"})[N]) := \text{Constant}(\text{StringLiteral}(S)); \curvearrowright \text{incomingArguments-aux}(L, N + \text{Int } 1)}$$

RULE

$$\frac{\text{incomingArguments-aux}(\bullet, N)}{(\text{Identifier}(\text{"\#incomingArgumentsArray"})[N]) := \text{NullPointer};}$$

SYNTAX $K ::= \text{syntaxNat}(Nat)$

RULE **SYNTAXNAT**

$$\frac{\text{syntaxNat}(N)}{\text{NoSuffix}(\text{DecimalConstant}(N))}$$

SYNTAX $K ::= \text{pgmArgs}(List\{K\})$
 $\quad | \text{argName}(List\{K\})$

DEFINE

$$\frac{\text{argName}(L)}{\text{Name}(\text{Identifier}(\text{"\#incomingArgumentsArray"}), \text{PointerType}(\text{ArrayType}(\text{JustBase}, \text{syntaxNat}((\text{length}_{ListK}(L)) + \text{Int } 1), \text{Specifier}(\text{List}(\bullet))))))}$$

DEFINE

$$\frac{\text{pgmArgs}(L)}{\text{DeclarationDefinition}(\text{InitNameGroup}(\text{Specifier}(\text{List}(\text{Char})), \text{List}(\text{InitName}(\text{argName}(L), \text{NoInit}))))}$$

RULE **VAL-NOINPUT**

$$\frac{\text{eval}(K)}{\text{eval}(K, \bullet, \text{"", } 0)}$$

These helpers are used to get around a bug in \mathbb{K} related to successive “/”s in strings.

SYNTAX $K ::= \text{stdinStr}$
 $\quad | \text{stdoutStr}$

DEFINE

stdinStr

("stdin:" + String "/") + String "/"

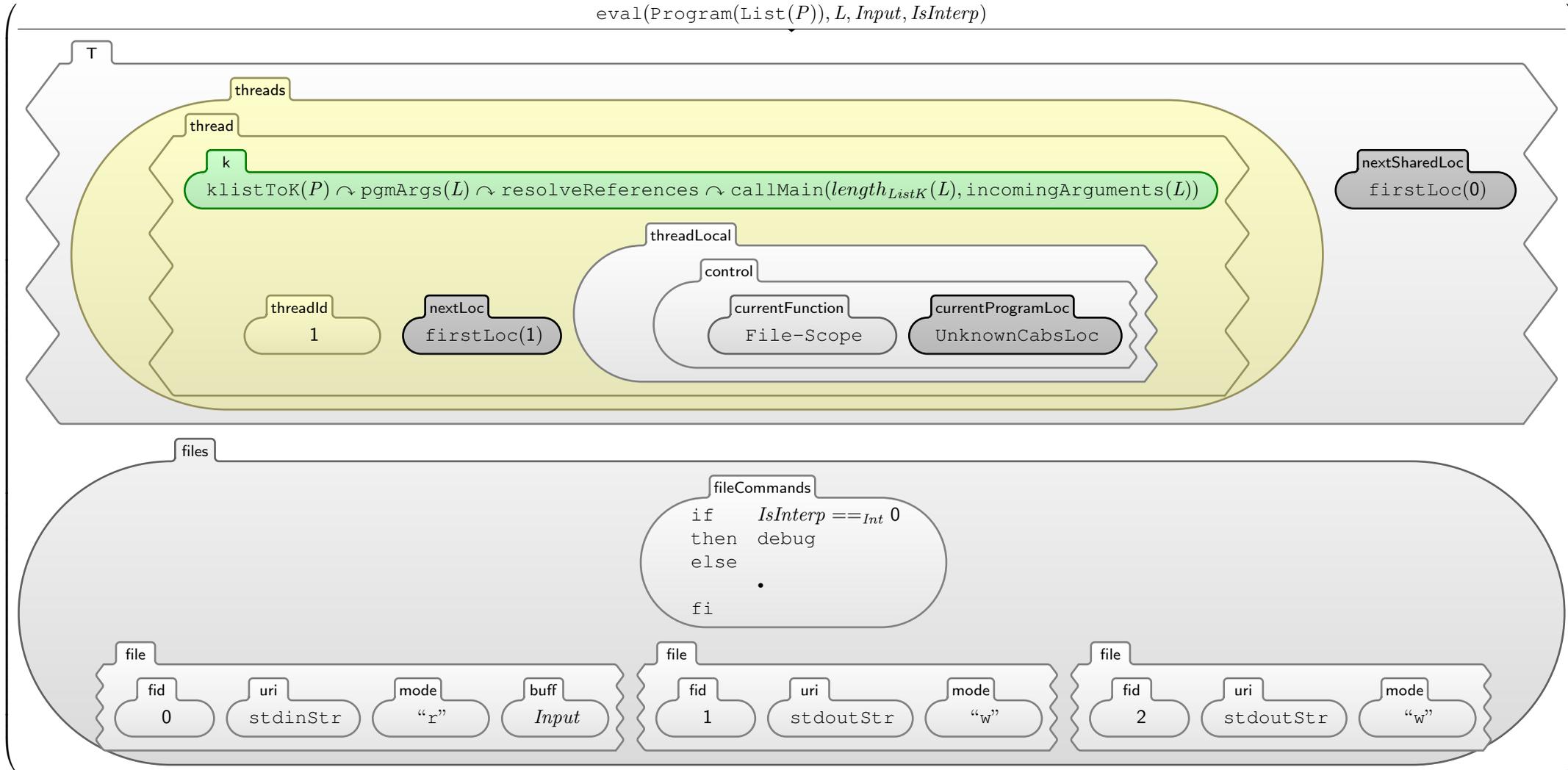
DEFINE

stdoutStr

("stdout:" + String "/") + String "/"

RULE EVAL-INPUT

eval(Program(List(*P*)), *L*, *Input*, *IsInterp*)



(n1570) §5.1.2.2.1 ¶1 The function called at program startup is named `main`. The implementation declares no prototype for this function. It shall be defined with a return type of `int` and with no parameters:

```
int main(void) { ... }
```

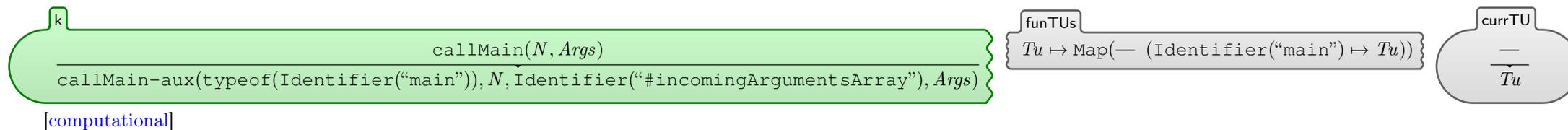
or with two parameters (referred to here as `argc` and `argv`, though any names may be used, as they are local to the function in which they are declared):

```
int main(int argc, char *argv[]) { ... }
```

or equivalent; or in some other implementation-defined manner.

this bit of indirection is used to check that the main prototype is correct, and to call it with the appropriate arguments

RULE CALL-MAIN



RULE

$$\frac{\text{callMain-aux}(t(\bullet, \text{functionType}(t(\bullet, \text{int}), \text{typedDecl}(t(\bullet, \text{void}), \text{---})), N, X, \text{---}))}{\text{Call}(\text{Identifier}(\text{"main"}), \text{List}(\bullet))}$$

RULE

$$\frac{\text{callMain-aux}(t(\bullet, \text{functionType}(t(\bullet, \text{int}), \bullet)), N, X, \text{---})}{\text{Call}(\text{Identifier}(\text{"main"}), \text{List}(\bullet))}$$

RULE

$$\frac{\text{callMain-aux}(t(\bullet, \text{functionType}(t(\bullet, \text{int}), \text{typedDecl}(t(\bullet, \text{int}), \text{---}), \text{typedDecl}(t(\bullet, \text{incompleteArrayType}(t(\bullet, \text{pointerType}(T))), \text{---})), N, X, \text{Args}))}{\text{Args} \curvearrowright \text{Call}(\text{Identifier}(\text{"main"}), \text{List}(N, X))}$$

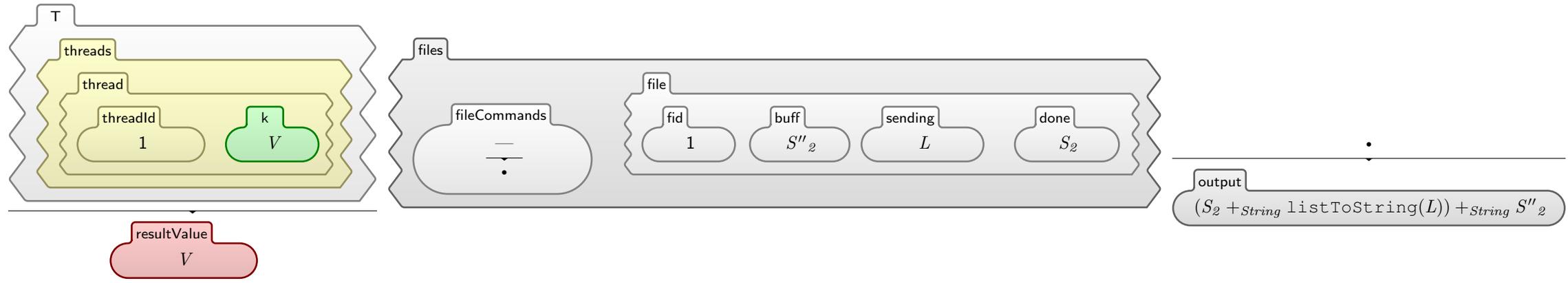
when $T ==_K t(\bullet, \text{char})$

RULE

$$\frac{\text{callMain-aux}(t(\bullet, \text{functionType}(t(\bullet, \text{int}), \text{typedDecl}(t(\bullet, \text{int}), \text{---}), \text{typedDecl}(t(\bullet, \text{pointerType}(t(\bullet, \text{pointerType}(T))), \text{---})), N, X, \text{Args}))}{\text{Args} \curvearrowright \text{Call}(\text{Identifier}(\text{"main"}), \text{List}(N, X))}$$

when $T ==_K t(\bullet, \text{char})$

RULE **TERMINATE**



[computational]

END MODULE

References

- [1] M. Acton. Understanding strict aliasing, June 2006. URL <http://cellperformance.beyond3d.com/articles/2006/06/understanding-strict-aliasing.html>. Accessed June 8, 2012. 103
- [2] J. Alves-Foss and F. S. Lam. Dynamic denotational semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 201–240. Springer, 1999. 14
- [3] D. J. Andrews, A. Garg, S. P. Lau, and J. R. Petchers. The formal definition of Modula-2 and its associated interpreter. In *2nd VDM-Europe Symposium (VDM'88)*, pages 167–177. Springer, 1988. 19
- [4] M. Anlauff. Xasm—an extensible, component-based abstract state machines language. In *International Workshop on Abstract State Machines*, Lecture Notes in Computer Science, pages 69–90. Springer-Verlag, 2000. 21
- [5] Apple Inc. Blocks programming topics, March 2011. URL <http://developer.apple.com/library/ios/documentation/cocoa/Conceptual/Blocks/Blocks.pdf>. 103
- [6] P. Arcaini, A. Gargantini, and E. Riccobene. CoMA: Conformance monitoring of Java programs by abstract state machines. In S. Khurshid and K. Sen, editors, *2nd International Conference on Runtime Verification (RV'11)*, volume 7186 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2011. 21
- [7] I. M. Asavoae and M. Asavoae. Collecting semantics under predicate abstraction in the K framework. In P. C. Ölveczky, editor, *Rewriting Logic and Its Applications*, volume 6381 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2010. Revised selected papers from the 8th International Workshop (WRLA'10). 3
- [8] I. Attali, D. Caromel, and M. Russo. A formal executable semantics for Java. In *Princeton University*, 1990. 14

- [9] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In T. Ball and M. Sagiv, editors, *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 55–66. ACM, 2011. 45, 46
- [10] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In J. Field and M. Hicks, editors, *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 509–520. ACM, 2012. 45
- [11] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press, 1989. 20
- [12] P. E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Brigham Young University, February 1998. 12
- [13] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. 10, 11, 12, 30, 32, 47, 53, 105
- [14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996. 103
- [15] M. Bofinger. *Reasoning about C programs*. PhD thesis, University of Queensland, February 1998. 12
- [16] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995. 16
- [17] E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer, 1999. 14
- [18] E. Börger, G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336: 2–3, 2003. 15, 21
- [19] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *PSDE'88*, pages 14–24. ACM Press, 1988. 2, 20
- [20] M. Bortin, C. Lüth, and D. Walter. A certifiable formal semantics of C. In T. Uustalu, J. Vain, and J. Ernits, editors, *20th Nordic Workshop on Programming Theory NWPT 2008*, Technical Report, pages 19–21. Institute of Cybernetics, Tallinn University of Technology, November 2008. 12

- [21] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 2nd edition, 1998. 9
- [22] D. Brown and D. Watt. JAS: A Java action semantics. In P. D. Mosses and D. A. Watt, editors, *2nd International Workshop on Action Semantics (AS'99)*, pages 43–56. Dept of Computer Science, University of Aarhus, 1999. Unrefereed paper. 14
- [23] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *9th International Working Conference on Source Code Analysis and Manipulation, SCAM'09*, pages 123–124. IEEE, 2009. 95
- [24] S. C. Cater and J. K. Huggins. An ASM dynamic semantics for Standard ML. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *International Workshop on Abstract State Machines, Theory and Applications (ASM'00)*, pages 203–222. Springer-Verlag, 2000. 16
- [25] CEA-LIST and INRIA-Saclay. Frama-C website, 2011. URL <http://frama-c.com/>. 19
- [26] F. Chalub and C. Braga. Maude MSOS tool. *Electronic Notes in Theoretical Computer Science*, 176(4):133–146, 2007. 2, 20, 21
- [27] F. Chen and G. Roşu. Rewriting logic semantics of Java 1.4, 2004. URL http://fs1.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java. 2, 14
- [28] J. Cheney and C. Urban. α Prolog: A logic programming language with names, binding and α -equivalence. In *20th International Conference on Logic Programming (ICLP'04)*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004. 21
- [29] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. 3
- [30] 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 *Lecture Notes in Computer Science*. Springer, 2007. 3, 20, 24, 55
- [31] M. Comstedt. Natural semantics specification for Java. Master's thesis, Linköping University, 2009. 14
- [32] J. V. Cook and S. Subramanian. A formal semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, November 1994. 9, 12

- [33] J. V. Cook, E. L. Cohen, and T. S. Redmond. A formal denotational semantics for C. Technical Report 409D, Trusted Information Systems, September 1994. 9, 11, 12, 47
- [34] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. In G. Hutton and A. P. Tolmach, editors, *14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pages 281–286. ACM Press, 2009. 19, 30, 105
- [35] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In *4th NASA Formal Methods Symposium (NFM 2012)*, April 2012. 95
- [36] C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Mapping modular SOS to rewriting logic. In M. Leuschel, editor, *12th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'02)*, volume 2664 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002. 20
- [37] P. Deransart and G. Ferrand. An operational formal definition of PROLOG: A specification method and its application. *New Gen. Comput.*, 10(2):121–171, April 1992. 13, 16
- [38] A. V. Deursen, J. Heering, H. A. D. Jong, M. D. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In *10th International Conference on Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2001. 20
- [39] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *34th International Conference on Software Engineering (ICSE'12)*, 2012. To appear. 63
- [40] A. Dijkstra and S. Swierstra. Ruler: Programming type rules. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 30–46. Springer Berlin / Heidelberg, 2006. 21
- [41] M. Dominus. Undefined behavior in Perl and other languages, October 2007. URL <http://blog.plover.com/prog/perl/undefined.html>. 71

- [42] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. H. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. World Wide Web Consortium, Recommendation REC-xquery-semantics-20070123, January 2007. 18
- [43] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82. Springer, 1999. 14
- [44] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited, 2000. 13, 14
- [45] 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. 53
- [46] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *4th International Workshop on Rewriting Logic and Its Applications (WRLA’02)*, volume 71 of *Electronic Notes in Theoretical Computer Science*, Amsterdam, September 2002. Elsevier. 3, 20, 59
- [47] C. Ellison and G. Roşu. A formal semantics of C with applications. Technical Report <http://hdl.handle.net/2142/17414>, University of Illinois, November 2010. 29
- [48] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In J. Field and M. Hicks, editors, *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’12)*, pages 533–544. ACM, 2012. 29, 71
- [49] C. Ellison and G. Roşu. Defining the undefinedness of C. Technical Report <http://hdl.handle.net/2142/30780>, University of Illinois, April 2012. 61
- [50] 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 *Lecture Notes in Computer Science*, pages 135–151, 2009. 3, 10, 102
- [51] J. Engblom. Dekker’s algorithm does not work, as expected, January 2008. URL <http://jakob.engbloms.se/archives/65>. 58
- [52] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1-2):71–103, January 2007. 21

- [53] 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 *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004. 2, 3, 14
- [54] K.-F. Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(5):295–357, July 2002. 18
- [55] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. 21
- [56] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. *Formal Methods and Software Engineering*, pages 15–29, 2004. 19
- [57] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, 2007. 19
- [58] Free Software Foundation. GNU compiler collection, 2010. URL <http://gcc.gnu.org>. 49
- [59] Free Software Foundation. Using the GNU compiler collection (GCC), 2010. URL <http://gcc.gnu.org/onlinedocs/gcc/>. 103
- [60] P. Fritzson, A. Pop, D. Broman, and P. Aronsson. Formal semantics based translator generation and tool development in practice. In *20th Australian Software Engineering Conference (ASWEC'09)*, pages 256–266. IEEE, 2009. 2, 21
- [61] FSF. C language testsuites: “C-torture” version 4.4.2, 2010. URL <http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html>. 51
- [62] E. R. Gansner and J. H. Reppy. The Standard ML basis library, 2004. URL <http://www.standardml.org/Basis/>. 71
- [63] A. Gargantini, E. Riccobene, and P. Scandurra. Model-driven language engineering: The ASMETA case study. In *3rd International Conference on Software Engineering Advances (ICSEA'08)*, pages 373–378. IEEE Computer Society, 2008. doi: 10.1109/ICSEA.2008.62. URL <http://dx.doi.org/10.1109/ICSEA.2008.62>. 21
- [64] A. Garrido, J. Meseguer, and R. Johnson. Algebraic semantics of the c preprocessor and correctness of its refactorings. Technical Report UIUCDCS-R-2006-2688, University of Illinois, February 2006. URL <http://hdl.handle.net/2142/11162>. 20
- [65] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005. 71

- [66] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and validation methods*, pages 9–36. Oxford University Press, Inc., 1995. 21
- [67] Y. Gurevich and J. K. Huggins. The semantics of the C programming language. In *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308, 1993. 8, 12
- [68] Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, October 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.06.017. 21
- [69] K. Hammond and C. Hall. A dynamic semantics for Haskell, 1993. Draft. 18
- [70] D. R. Hanson and C. W. Fraser. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. 53
- [71] C. Hathhorn. Semantics for CUDA in K, 2012. URL <https://github.com/chathhorn/cuda-sem>. 103
- [72] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. Technical report, CWI (Centre for Mathematics and Computer Science), 1999. 2
- [73] M. Hills and G. Roşu. KOOL: An application of rewriting logic to language prototyping and analysis. In *18th International Conference on Rewriting Techniques and Applications (RTA '07)*, volume 4533 of *Lecture Notes in Computer Science*, pages 246–256. Springer, 2007. 2
- [74] M. Hills, F. Chen, and G. Roşu. A rewriting logic approach to static checking of units of measurement in C. In *9th International Workshop on Rule-Based Programming (RULE'08)*, volume To Appear of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2008. 3
- [75] M. Holmén. Natural semantics specification and frontend generation for Java 1.2. Master’s thesis, Linköping University, 2009. 14
- [76] IEEE. Posix.1c, threads extensions. Technical Report IEEE Std 1003.1c-1995, IEEE, 1996. 23, 43
- [77] INRIA. CompCert C compiler, version 1.9, August 2011. URL <http://compcert.inria.fr/>. 11
- [78] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:1990: Programming languages—C. Technical report, International Organization for Standardization, 1990. 23

- [79] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:1999: Programming languages—C. Committee Draft N1256, International Organization for Standardization, December 1999. 23, 29, 34, 82, 83
- [80] 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. 3, 23, 64, 104
- [81] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:2011: Programming languages—C. Committee Draft N1570, International Organization for Standardization, August 2011. 5, 6, 13, 23, 24, 29, 33, 35, 37, 38, 40, 41, 42, 43, 44, 46, 55, 62, 63, 64, 65, 66, 67, 68, 70, 73, 75, 77, 79, 80, 81, 82, 94, 97, 98, 99, 103
- [82] ISO/IEC JTC 1, SC 22, WG 14. C secure coding rules. Committee Draft N1579, International Organization for Standardization, September 2011. 94
- [83] 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>. 22, 104
- [84] S. L. P. Jones and P. Wadler. A static semantics for Haskell. Technical report, Department of Computing Science, University of Glasgow, 1992. 18
- [85] H. Jula and N. G. Fruja. An executable specification of C#. In *Abstract State Machines*, pages 275–288, 2005. 15
- [86] G. Kahn. Natural semantics. In *STACS'87*, pages 22–39. Springer, 1987. 20
- [87] R. Kelsey, W. Clinger, and J. Rees. Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33:26–76, 1998. 71
- [88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1978. 8, 22, 53
- [89] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Ralfkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In J. Field and M. Hicks, editors, *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 285–296. ACM, 2012. 21
- [90] M. Kulaš and C. Beierle. Defining Standard Prolog in rewriting logic. In K. Futatsugi, editor, *Proc. of the 3rd Int. Workshop on Rewriting Logic and its Applications (WRLA'2000), Kanazawa*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001. 16, 20

- [91] M. R. Lakin. *An executable meta-language for inductive definitions with binders*. PhD thesis, University of Cambridge, 2010. 21
- [92] C. Lattner. What every C programmer should know about undefined behavior, May 2011. URL <http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html>. 63
- [93] C. Lattner. LLVM assembly language reference manual, February 2012. URL <http://llvm.org/docs/LangRef.html>. 71
- [94] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. 62, 69
- [95] S. Maharaj and E. Gunter. Studying the ML module system in HOL. In *Higher Order Logic Theorem Proving and its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 346–361. Springer-Verlag, 1994. 13, 15
- [96] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. 3
- [97] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *15th International Conference on Rewriting Techniques and Applications (RTA'04)*, Lecture Notes in Computer Science, pages 301–311. Springer, 2004. 21
- [98] J. McCarthy. Datalog for PLT Scheme, 2010. URL <http://planet.racket-lang.org/package-source/jaymccarthy/datalog.plt/1/3/>. 21
- [99] M. Mehlich. CheckPointer—A C memory access validator. In *11th International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*, pages 165–172. IEEE, 2011. 95
- [100] P. Meredith, M. Hills, and G. Roşu. An executable rewriting logic semantics of K-Scheme. In D. Dube, editor, *2007 Workshop on Scheme and Functional Programming (SCHEME'07)*, Technical Report DIUL-RT-0701, pages 91–103. Laval University, 2007. 2, 17
- [101] P. Meredith, M. Hills, and G. Roşu. A K definition of Scheme. Technical Report Department of Computer Science UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign, 2007. 17
- [102] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of Verilog. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*. IEEE, 2010. To appear. 2, 17

- [103] M. Mernik, M. Lenic, E. Avdicausevic, and V. Zumer. Compiler/interpreter generator system LISA. In *33rd Hawaii International Conference on System Sciences (HICSS'00)*, pages 590–594. IEEE, 2000. 20
- [104] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. 3, 20, 24, 30
- [105] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007. 20
- [106] J. Meseguer and G. Roşu. The rewriting logic semantics project: A progress report. In *17th International Symposium on Fundamentals of Computation Theory (FCT'11)*, volume 6914 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2011. Invited talk. 20
- [107] M. Might. Abstract interpreters for free. In *17th International Conference on Static Analysis (SAS'10)*, pages 407–421. Springer, 2010. 2
- [108] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. 15
- [109] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 13, 15
- [110] MISRA Consortium. MISRA-C: 2004—Guidelines for the use of the C language in critical systems. Technical report, MIRA Ltd., October 2004. 94
- [111] MITRE Corporation. The common weakness enumeration (CWE) initiative, 2012. URL <http://cwe.mitre.org/>. 94
- [112] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004. 20
- [113] T. Nagel. Troubles with GCC signed integer overflow optimization, January 2010. URL <http://thiemonagel.de/2010/01/signed-integer-overflow/>. 65
- [114] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *11th International Conference on Compiler Construction (CC'02)*, *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002. 30, 178

- [115] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. 95
- [116] NIST. Juliet test suite for C/C++, December 2010. URL <http://samate.nist.gov/SRD/testsuite.php>. 94
- [117] 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. 102
- [118] L. C. Noll, S. Cooper, P. Seebach, and L. A. Broukhis. The international obfuscated C code contest, 2010. URL <http://www.ioccc.org/>. 53
- [119] M. Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, December 1998. 9, 10, 11, 12, 62, 63, 78, 89
- [120] 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. 10, 13, 15
- [121] M. Ouimet and K. Lundqvist. The TASM toolset: Specification, simulation, and formal verification of real-time systems. In *19th International Conference on Computer Aided Verification (CAV'07)*, pages 126–130, Berlin, Heidelberg, 2007. Springer-Verlag. 21
- [122] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *22th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009. 46
- [123] G. J. Pace and J. He. Formal reasoning with Verilog HDL. In *In Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand*, 1998. 17
- [124] N. S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998. 10
- [125] N. S. Papaspyrou. Denotational semantics of ANSI C. *Computer Standards and Interfaces*, 23(3):169–185, 2001. 10, 11, 12, 55
- [126] 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. 89
- [127] U. F. Pleban. Compiler prototyping using formal semantics. *SIGPLAN Not.*, 19:94–105, June 1984. 2

- [128] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60:60–61, 2004. 41
- [129] J. Regehr. A guide to undefined behavior in C and C++, July 2010. URL <http://blog.regehr.org/archives/213>. 63
- [130] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In J. Vitek, H. Lin, and F. Tip, editors, *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, pages 335–346. ACM, 2012. 61, 87, 88
- [131] E. Riccobene, P. Scandurra, and F. Albani. A modeling and executable language for designing and prototyping service-oriented applications. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:4–11, 2011. 21
- [132] A. Riesco, A. Verdejo, and N. Martí-Oliet. A complete declarative debugger for Maude. In *13th International Conference on Algebraic Methodology and Software Technology (AMAST'10)*, pages 216–225. Springer-Verlag, 2011. 3, 20
- [133] G. Roşu. K: A rewriting-based framework for computations—Preliminary version. Technical Report UIUCDCS-R-2007-2926, University of Illinois, Department of Computer Science, 2007. 3, 28
- [134] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. 2, 3, 24, 28
- [135] G. Roşu and A. Ştefănescu. Matching logic: A new program verification approach (NIER track). In *30th International Conference on Software Engineering (ICSE'11)*, pages 868–871, 2011. 102, 103
- [136] G. Roşu and A. Ştefănescu. From Hoare logic to matching logic reachability. In *18th International Symposium on Formal Methods (FM'12)*, Lecture Notes in Computer Science. Springer, 2012. To appear. 103
- [137] G. Roşu and A. Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In *39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, Lecture Notes in Computer Science. Springer, 2012. To appear. 103
- [138] G. Roşu, W. Schulte, and T. F. Şerbănuţă. Runtime verification of C memory safety. In *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, 2009. 2, 3, 32, 80

- [139] 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)*, volume 6486 of *Lecture Notes in Computer Science*, pages 142–162, 2010. 3, 10, 102, 103
- [140] Ruby Standardization WG. Programming languages—Ruby. Draft, Information-technology Promotion Agency, August 2010. 71
- [141] S. Sarkar, M. Batty, S. Owens, K. Memarian, L. Maranget, J. Alglave, P. Sewell, and D. Williams. Synchronising C/C++ and POWER. In J. Vitek, H. Lin, and F. Tip, editors, *33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, pages 311–322. ACM, 2012. 45
- [142] H. Sasaki. A formal semantics for Verilog-VHDL simulation interoperability by abstract state machine. In *2nd Conference on Design, Automation and Test in Europe (DATE'99)*. ACM, 1999. 13, 17
- [143] J. Schmid. *Introduction to AsmGofer*, March 2001. URL <http://www.tydo.de/download/Doktorarbeit/AsmGoferIntro.pdf>. 21
- [144] R. C. Seacord and J. A. Rafail. The CERT C secure coding standard, 2008. 94
- [145] J. R. Senning. Solution of the dining philosophers problem using shared memory and semaphores, January 2000. URL <http://www.math-cs.gordon.edu/courses/cs322/projects/p2/dp/>. 59
- [146] T. F. Şerbănuţă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois, 2010. URL <http://hdl.handle.net/2142/18252>. 2, 3, 19, 28, 45, 76, 90
- [147] T. F. Şerbănuţă and G. Roşu. KRAM—extended report. Technical Report <http://hdl.handle.net/2142/17337>, UIUC, September 2010. 2
- [148] 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 *Lecture Notes in Computer Science*, pages 104–122, 2010. 2, 3, 24, 28
- [149] T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009. 19, 20
- [150] T. F. Şerbănuţă, A. Arusoaiu, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu. The K primer (version 2.5). In M. Hills, editor, *K'11*, *Electronic Notes in Theoretical Computer Science*, to appear. 2, 24, 50, 107

- [151] J. Sevčik, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In T. Ball and M. Sagiv, editors, *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*, pages 43–54. ACM, 2011. 45, 46
- [152] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semantacist. In *12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 1–12. ACM, 2007. 21
- [153] G. J. Smeding. An executable operational semantics for Python. Master's thesis, University of Utrecht, January 2009. 18
- [154] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised⁶ Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010. 17, 21
- [155] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, 2001. 14, 21
- [156] T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs. A linear operational semantics for termination and complexity analysis of ISO Prolog. In *21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11)*, LNCS. Springer, 2011. 17
- [157] S. Subramanian and J. V. Cook. Mechanical verification of C programs. In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, January 1996. 9
- [158] S. Summit. C programming FAQs: Frequently asked questions, 2005. URL <http://www.c-faq.com/>. 6
- [159] TIOBE Software BV. TIOBE programming community index, July 2010. URL <http://www.tiobe.com/index.php/content/paperinfo/tpci/>. 22
- [160] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The ASF+SDF compiler, 2000. 20
- [161] M. VanInwegen and E. Gunter. HOL-ML. In J. J. Joyce and C.-J. H. Seger, editors, *6th International Workshop on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, Lecture Notes in Computer Science, pages 61–74. Springer, 1994. 13, 15

- [162] C. Wallace. The semantics of the C++ programming language. In *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1993. 15
- [163] C. Wallace. The semantics of the Java programming language: Preliminary version. Technical Report CSE-TR-355-97, University of Michigan, 1997. 14
- [164] D. Watt. The static and dynamic semantics of Standard ML. In P. D. Mosses and D. A. Watt, editors, *2nd International Workshop on Action Semantics (AS'99)*, pages 155–172. Dept of Computer Science, University of Aarhus, 1999. 16
- [165] D. N. Welton. Programming language popularity, April 2011. URL <http://langpop.com/>. Accessed May 23, 2012. 22
- [166] M. Wolczko. Semantics of Smalltalk-80. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *1st European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 1987. 13, 16
- [167] X3J11 Technical Committee on the C Programming Language under project 381-D by American National Standards Committee on Computers and Information Processing (X3). Programming language C. Technical Report X3.159-1989, ANSI, 1989. 22
- [168] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, pages 283–294, 2011. 7, 87
- [169] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In J. Field and M. Hicks, editors, *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 427–440. ACM, 2012. 13, 16
- [170] W. Zimmermann and A. Dold. A framework for modeling the semantics of expression evaluation with abstract state machines. In *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 391–406, 2003. 9