

# A Calculus for Unreachable Code

PETER ZHONG, PLT, Northwestern University, U.S.A.

SHU-HUNG YOU, PLT, Northwestern University, U.S.A.

SIMONE CAMPANONI, Northwestern University, U.S.A.

ROBERT BRUCE FINDLER, PLT, Northwestern University, U.S.A.

MATTHEW FLATT, University of Utah, U.S.A.

CHRISTOS DIMOULAS, PLT, Northwestern University, U.S.A.

In Racket, the LLVM IR, Rust, and other modern languages, programmers and static analyses can hint, with special annotations, that certain parts of a program are unreachable. Same as other assumptions about undefined behavior; the compiler assumes these hints are correct and transforms the program aggressively.

While compile-time transformations due to undefined behavior often perplex compiler writers and developers, we show that the essence of transformations due to unreachable code can be distilled in a surprisingly small set of simple formal rules. Specifically, following the well-established tradition of understanding linguistic phenomena through calculi, we introduce the first calculus for unreachable. Its term-rewriting rules that take advantage of unreachable fall into two groups. The first group allows the compiler to delete any code downstream of unreachable, and any effect-free code upstream of unreachable. The second group consists of rules that eliminate conditional expressions when one of their branches is unreachable. We show the correctness of the rules with a novel logical relation, and we examine how they correspond to transformations due to unreachable in Racket and LLVM.

## ACM Reference Format:

Peter Zhong, Shu-Hung You, Simone Campanoni, Robert Bruce Fidler, Matthew Flatt, and Christos Dimoulas. 2024. A Calculus for Unreachable Code. 1, 1 (July 2024), 26 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 THE FLEETING ESSENCE OF UNREACHABLE

Modern compilers routinely take advantage of undefined behavior. Specifically, they assume that a program never performs operations that exhibit undefined behavior, and then they use this assumption to optimize the program. After all, a program that lives outside the confines of defined behavior is meaningless. Hence, the effect of optimizations on its behavior is a moot point. While the use of undefined behavior in semantics is controversial, and with good reason (CVE-2014-0160 2014), undefined behavior is widely used and, as Jung (2021) eloquently argued, not all forms of undefined behavior are the same.

In this paper, we attempt to bring a sound semantics footing of a particular form of undefined behavior: unreachable. Programmers and tools that transform code, such as static analyses, introduce the construct unreachable in programs to claim that some part of the program can

---

Authors' addresses: Peter Zhong, PLT, Northwestern University, U.S.A., [peterzhong2023@u.northwestern.edu](mailto:peterzhong2023@u.northwestern.edu); Shu-Hung You, PLT, Northwestern University, U.S.A., [shu-hung.you@eecs.northwestern.edu](mailto:shu-hung.you@eecs.northwestern.edu); Simone Campanoni, Northwestern University, U.S.A., [simonec@eecs.northwestern.edu](mailto:simonec@eecs.northwestern.edu); Robert Bruce Fidler, PLT, Northwestern University, U.S.A., [robby@cs.northwestern.edu](mailto:robby@cs.northwestern.edu); Matthew Flatt, University of Utah, U.S.A., [mflatt@cs.utah.edu](mailto:mflatt@cs.utah.edu); Christos Dimoulas, PLT, Northwestern University, U.S.A., [chrdimo@northwestern.edu](mailto:chrdimo@northwestern.edu).

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2024 Association for Computing Machinery.

XXXX-XXXX/2024/7-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

<pre>(define (func x)   (if (pair? x)       (car x)       (unsafe-assert-unreachable)))</pre>	<pre>define i32 @func(i32 %0) { 1: %2 = icmp sgt i32 %0, 0    br i1 %2, label %3, label %4 3: ret i32 0           ; preds = %1 4: unreachable       ; preds = %1 }</pre>
---	--

Fig. 1. Two Flavors of unreachable: Racket (Left) and LLVM IR (Right)

never be evaluated. Concretely, figure 1 depicts two code snippets that demonstrate unreachable in two different linguistic contexts. In the Racket snippet, `unsafe-assert-unreachable` signals to the Racket compiler that the else-branch of the `if` expression can never be reached. Hence, `unsafe-assert-unreachable` gives the Racket compiler the freedom to optimize away the entire `if` expression, effectively asserting that `x` is always a pair. In the same spirit, LLVM’s `unreachable` instruction informs the compiler that the control-flow of a program never reaches that instruction. Therefore, in the LLVM IR snippet in figure 1, all three basic blocks can be collapsed into one. In other words, same as for other forms of undefined behavior, `unreachable` describes an assumption about the evaluation of a program that a compiler can take advantage of while optimizing the program. If the assumption is true, then the program is transformed to an equivalent, possibly more performant, new version. If the assumption is false, then the program is meaningless to begin with and the semantics of the language gives the compiler a free hand to produce any code whatsoever.

The discussion so far suggests that the phrase “the meaning of unreachable” is an oxymoron. However, we show that staple tools of PL semanticists, such as calculi and equational theories, can faithfully describe the interplay between unreachable, compile-time transformations and program behavior, and hence, reveal the essence of unreachable through the way compilers use it.

Standing on the shoulders of fifty years of PL tradition (Plotkin 1975), we turn to the  $\lambda$  calculus and we construct the first calculus that captures the essence of unreachable. In our unreachable calculus, we describe the legal basic steps that a compiler performs when transforming a program due to unreachable as local term-rewriting rules. At a high level, there are two groups of rules: those that eliminate unreachable and those that propagate it through a program. The first group of rules dictates how a compiler can discard computations around unreachable and these are the same as rules that discard computations around expressions that are guaranteed to signal errors. The second group describes how a compiler can leverage unreachable to eliminate control-flow paths that cannot be reached. This group consists of just a single rule and its symmetric counterpart, showing how `if` expressions and `unreachable` interact.

There are two questions about the unreachable calculus. The first question is correctness. Specifically, do the rules of the calculus preserve program semantics — as induced by the standard reduction of the calculus — under the assumption that evaluation of the program never reaches unreachable? The answer is affirmative but the proof is subtle because the assumption about unreachable is not compositional. Even when it is true that a particular program does not reach unreachable no matter its input, there may be subexpressions that, if they were to be evaluated directly, would reach unreachable. Worse, the guarantee that ensures the entire program cannot reach unreachable may be arbitrarily complex. We show how to handle this complexity via the construction of a novel logical relation, while keeping the rules of the calculus simple and intuitive.

The second question is about generality. Does the simplicity of the calculus keep it from capturing the optimizations in production compilers? To answer this question, we focus on the two compilers: Racket’s and LLVM. For Racket, we performed an audit of the Racket codebase, which revealed that all of the rewrites that Racket performs because of unreachable map to the rules of our calculus.

For LLVM, we cannot draw a similarly direct conclusion because LLVM is too different from our  $\lambda$ -based calculus. Instead, inspired by an audit of the LLVM codebase and additions to LLVM codebase at the same time as the addition of unreachable we design a transformation for a subset of LLVM IR (technically `Vminus` (Zhao et al. 2013)) that captures some of the ways that LLVM exploits unreachable. To do so, we compose our transformation with a version of Kelsey (1995)’s functions that translate SSA to and from the  $\lambda$  calculus.

Overall, this paper identifies unreachable as an undefined behavior whose semantics can, surprisingly, be captured via a simple set of rules. Even better, these rules match how compiler writers conceptualize optimization as local, context-insensitive rewrites. In short, we see this paper as a first step towards a systematic demystification of undefined behavior.

The remainder of the paper is organized as follows. Section 2 explains the essence of unreachable, and of our calculus, through examples. Section 3 presents the unreachable calculus while section 4 proves its correctness. Section 5 discusses the extension of the calculus with arbitrary semantics-preserving rules that are useful when proving equivalences with the calculus. Section 6 shows how our calculus can shed light to other forms of undefined behavior beyond unreachable as evidence for our approach’s value in understanding undefined behavior. Sections 7 and 8 connect our calculus to Racket and LLVM, respectively. The last two sections discuss related work and conclude.

## 2 THE ESSENCE OF UNREACHABLE, BY EXAMPLE

From a compiler engineering perspective, the value of unreachable is that it contributes to the clean separation between the different passes of the compilation process. For example, a static analysis pass can deduce that some part of a program is unreachable, and mark it as such with unreachable to facilitate a subsequent pass that aims to optimize the program. Given that the ability of compilers to reliably detect unreachable code is limited, programmers can also take advantage of the same mechanism to share with the compiler facts about whether some parts of a program are unreachable. Once the programmer (or the analysis) has communicated that some code is unreachable, the compiler can take advantage of this information in various different ways.

The simplest such way is by treating unreachable as erroneous code, or more generally, as code that never returns. Specifically, the compiler can erase terminating, effect-free computations that happens before unreachable, and all computation that is guaranteed to happen after it. For example, the Racket compiler can simplify the expression

```
(begin (+ x 1)
       (unsafe-assert-unreachable)
       (+ y 2))
```

to just `(unsafe-assert-unreachable)`, where `(unsafe-assert-unreachable)` is a syntactic form that programmers use to declare that some part of an expression is unreachable.<sup>1</sup>

Treating unreachable as erroneous code, however, misses optimization opportunities because it ignores the semantics of unreachable code. To see how, consider the following Racket code:

```
(define (m-dist p)
  (match p
    [(cons x y) (+ (abs x) (abs y))]))
```

The `m-distance` function computes the Manhattan distance for a point represented as a pair of (real) numbers. Its body consists of use of the `match` which roughly expands to a conditional and appropriate uses of accessor functions:

<sup>1</sup>Racket also offers `assert-unreachable`, which is guaranteed to throw an error when evaluated. Similar to Racket, Rust also has two variants, one that allows aggressive optimization and one that is safe, for testing and debugging.

```
(define (m-dist p)
  (if (pair? p)
      (+ (abs (car x)) (abs (cdr x)))
      (error 'match-failure)))
```

In general the two accessor functions in the above snippet, `car` and `cdr`, come with checks that defensively inspect their argument to make sure it is a pair. However, the Racket compiler can deduce that the uses of the accessor functions in the snippet are guarded by the `pair?` (on a variable that isn't modified). Hence, the compiler replaces them with their unsafe variants:

```
(define (m-dist p)
  (if (pair? p)
      (+ (abs (unsafe-car p)) (abs (unsafe-cdr p)))
      (error 'match-failure)))
```

And that's how far the Racket compiler can go on its own. However, if the author of the code knows that `m-dist` is only applied to points, i.e., pairs, then, they can convey this information to the Racket compiler by adding to the `match` expression in the body of `m-dist` a catch-all case that uses `unsafe-assert-unreachable`, the Racket variant of `unreachable` that has undefined behavior, giving the compiler the license to *assume* that it is never reached. Rust has a similar construct, `unreachable_unchecked`, which also has undefined behavior if it is ever reached.

```
(define (m-dist p)
  (match p
    [(cons x y) (+ (abs x) (abs y))]
    [_ (unsafe-assert-unreachable)]))
```

As in the steps as above, the Racket compiler simplifies the body of `m-dist` to:

```
(define (m-dist p)
  (if (pair? p)
      (+ (abs (unsafe-car p)) (abs (unsafe-cdr p)))
      (unsafe-assert-unreachable)))
```

Since, the else-branch of the conditional is marked as `unreachable`, the Racket compiler can assume that in a program that uses `m-dist`, `m-dist` consumes only pairs. Hence, the compiler can transform the `m-dist` to eliminate the `unreachable` branch and replace it with a `begin` expression:

```
(define (m-dist p)
  (begin (pair? p)
         (+ (abs (unsafe-car p)) (abs (unsafe-cdr p)))))
```

In short, the Racket compiler applies one of the fundamental rules of `unreachable`:

```
(if e_1 e_2 (unsafe-assert-unreachable))
=
(begin e_1 e_2)
```

The use of the rule unlocks one more opportunity for simplifying (`m-dist`). Since `pair?` has no side effects, the Racket compiler drops it to obtain the final version of `m-dist`:

```
(define (m-dist p)
  (+ (abs (unsafe-car p)) (abs (unsafe-cdr p))))
```

$$\begin{aligned}
e ::= & x \mid c \mid \lambda x.e \mid op\ e\ e \mid e\ e \mid (if\ e\ e\ e) \mid (begin\ e\ e) \mid unreachable \mid error_k \\
c ::= & n \mid false \mid true \quad n \in \text{numbers} \quad op \in \text{primitive operations}
\end{aligned}$$

Fig. 2. The Syntax of the unreachable Calculus

$$\begin{array}{c}
\boxed{e \rightarrow_s e} \quad \boxed{e R_s e} \\
\\
E ::= [] \mid op\ E\ e \mid op\ v\ E \mid (if\ E\ e\ e) \mid E\ e \mid v\ E \mid (begin\ E\ e) \\
v ::= c \mid \lambda x.e \quad a ::= c \mid \lambda x.e \mid unreachable \mid error_k \\
\\
\frac{}{(if\ false\ e_1\ e_2)\ R_s\ e_2} \text{ S.1} \quad \frac{v \neq false}{(if\ v\ e_1\ e_2)\ R_s\ e_1} \text{ S.2} \quad \frac{}{((\lambda x.e)\ v)\ R_s\ e[v/x]} \text{ S.3} \\
\frac{}{(begin\ v\ e)\ R_s\ e} \text{ S.4} \quad \frac{\delta(op, v_1, v_2) \equiv c}{(op\ v_1\ v_2)\ R_s\ c} \text{ S.5} \quad \frac{e\ R_s\ e'}{E[e] \rightarrow_s E[e']} \text{ S.6} \\
\frac{E \neq []}{E[unreachable] \rightarrow_s unreachable} \text{ S.7} \quad \frac{E \neq []}{E[error_k] \rightarrow_s error_k} \text{ S.8} \\
\frac{v_1 \neq \lambda x.e}{(v_1\ v_2)\ R_s\ error_\beta} \text{ S.9} \quad \frac{(op, v_1, v_2) \notin \text{dom}(\delta)}{(op\ v_1\ v_2)\ R_s\ error_\delta} \text{ S.10}
\end{array}$$

Fig. 3. The Standard Reduction of the unreachable Calculus

Surprisingly, the two examples in this section are sufficient to describe the full essence of the compile-time treatment of unreachable: either compilers treat unreachable same as an error (first example), or as the justification for simplifying the branches of conditionals (second example). The following section gives a formal account of this insight with the unreachable calculus and its compile-time reductions.

### 3 THE ESSENCE OF UNREACHABLE, FORMALLY

Our calculus extends the call-by-value  $\lambda$  calculus with a construct unreachable and unreachable-specific term rewriting rules. Specifically, there are two groups of such rules: the first, which extends the standard reduction of the call-by-value  $\lambda$  calculus, corresponds to the run time behavior of unreachable; the second corresponds to legal compile-time transformations of unreachable.

#### 3.1 The Unreachable Calculus and its Standard Reduction

Figure 2 presents the syntax of the unreachable calculus. Expressions consist of variables, constants,  $\lambda$  expressions, binary operators, conditionals, sequencing, and a family of different errors (indexed by  $k$ ). The only unique syntactic element is unreachable, which represents annotations that indicate unreachable code like Racket's `unsafe-assert-unreachable`.

Figure 3 describes the standard reduction of the calculus. It starts with a standard definition of evaluation contexts that ensure a left-to-right order of evaluation. The  $v$  non-terminal describes values and  $a$  final answers. The remainder of the figure defines the notions of reduction  $R_s$ , whose

$$\begin{array}{c}
\frac{e \rightarrow_p e'}{e \rightarrow_c e'} \quad \frac{e \rightarrow_u e'}{e \rightarrow_c e'} \quad \boxed{e \rightarrow_c e} \\
C ::= [] \mid op C e \mid op e C \mid (if C e e) \mid (if e C e) \mid (if e e C) \\
\mid C e \mid e C \mid \lambda x. C \mid (begin C e) \mid (begin e C)
\end{array}$$

Fig. 4. The Compile-time Relation (and the definition of contexts)

compatible closure over evaluation contexts is the core of the standard reduction  $\rightarrow_s$  (“s” for standard).

The notions of reduction of the calculus are mostly as expected. Similar to Racket, the unreachable calculus employs “truthy” values in its conditionals. Hence, rule S.1 reduces an if expression to its else-branch if the test position reduces to the value false. Any other value causes the conditional to reduce to its then-branch (rule S.2); we use  $\equiv$  for syntactic equality, here and throughout the paper. The  $R_s$  relation also includes  $\beta$ -value (rule S.3) and the usual sequencing rule that discards the first sub-expression of a begin expression when it is a value (rule S.4). Rule S.5 defers to the  $\delta$  function to handle calls to primitive operators.

Rules S.9 and S.10 introduce errors. When a function application involves a value other than a  $\lambda$  expression in the function position, it reduces to a specific error with the label  $\beta$ . Similarly, when a primitive operator encounters arguments that do not make sense, it reduces to an error with the label  $\delta$ . We equip the calculus with a family of errors in order to account for the common linguistic setting where there are multiple, semantically distinct types of side-effects besides non-termination.

Rules S.6 and S.8 form a typical definition of the standard reduction  $\rightarrow_s$  in a language with errors. Rule S.6 lifts the notions of reduction to evaluation contexts. Rule S.8 discards the evaluation context around an error to terminate the reduction.

Rule S.7 is the only rule that is specific to unreachable. It treats unreachable the same as an error. The rule is based on Rust’s and Racket’s safe variants of unreachable that are supposed to be used to check assumptions about code unreachability during debugging. After all, from the perspective of a compiler, which is the perspective we examine in this paper, programs that evaluate unreachable are plain wrong, and hence evaluating unreachable should result in some form of error. Furthermore, as we discuss further in section 4, this run time behavior of unreachable plays an important role in the compositional proof of the correctness of the compile-time transformations that the unreachable calculus captures.

### 3.2 Compile-Time Transformations as Reduction Relations

Figure 4 shows the relation  $\rightarrow_c$  (“c” for compile), which captures compile-time transformations due to unreachable. This relation does not define a program transformation algorithm that a compiler might use; instead it describes the set of valid basic transformations steps that a compiler is allowed to perform. In other words, the relation is a specification of how a correct compiler may take advantage of unreachable in order to simplify a program.

The  $\rightarrow_c$  relation is broken down into two pieces:  $\rightarrow_p$  (“p” for propagate), which captures transformations of unreachable that are legal for a safe variant of unreachable, namely when it behaves like an error, and  $\rightarrow_u$  (“u” for undefined), which captures the undefined behavior of unreachable, namely how it interacts with conditional expressions. Figure 4 also gives the definition of contexts, which are used in the definitions of both  $\rightarrow_p$  and  $\rightarrow_u$ .

Figure 5 presents  $\rightarrow_p$ . Rules P.1 to P.5 in figure 5 allow the compiler to propagate unreachable downstream and upstream in a compound expression. Specifically, rules P.2 and P.4 enable the

$$\begin{array}{c}
\frac{\text{SAFE}(e)}{(\text{begin } e \text{ unreachable}) R_p \text{ unreachable}} \text{ P.1} \quad \frac{\boxed{e R_p e} \quad \boxed{e \rightarrow_p e}}{(\text{begin unreachable } e) R_p \text{ unreachable}} \text{ P.2} \\
\frac{}{((\lambda x. \text{unreachable}) e) R_p (\text{begin } e \text{ unreachable})} \text{ P.3} \\
\frac{}{(\text{unreachable } e) R_p \text{ unreachable}} \text{ P.4} \quad \frac{}{(e \text{ unreachable}) R_p (\text{begin } e \text{ unreachable})} \text{ P.5} \\
\frac{e R_p e'}{C[e] \rightarrow_p C[e']} \text{ Ctx.P} \quad \frac{e' R_p e}{C[e] \rightarrow_p C[e']} \text{ CtxSym.P}
\end{array}$$

Fig. 5. The Compile-time Unreachable Propagation Relations

$$\begin{array}{c}
\frac{}{(\text{if } e_c \text{ unreachable } e_f) R_u (\text{begin } e_c e_f)} \text{ U.1} \quad \frac{\boxed{e R_u e} \quad \boxed{e \rightarrow_u e}}{(\text{if } e_c e_t \text{ unreachable}) R_u (\text{begin } e_c e_t)} \text{ U.2} \\
\frac{e R_u e'}{C[e] \rightarrow_u C[e']} \text{ Ctx.U}
\end{array}$$

Fig. 6. The Compile-time Unreachable Undefined Behavior Relations

compiler to eliminate all expressions that follow unreachable. In essence, they capture the idea that all computation downstream of unreachable code is also unreachable since computation never progresses past unreachable. Similarly, the compiler can eliminate all expressions that precede unreachable, as long as they are safe. Roughly, safe expressions are those that evaluate to a value, i.e., they have no side-effects:

*Definition 3.1 (safety).* An expression  $e$  is *safe*, written as  $\text{SAFE}(e)$ , if for all closing substitutions  $\vartheta$ , we have  $\vartheta(e) \rightarrow_s^* v$ .

Given this definition of safe expressions, rule P.1 embodies how a compiler benefits from the combination of safe expressions and unreachable. It allows unreachable to “eat” safe expressions upstream. After all, if the value-result of a safe expression is not used, then it can be discarded without affecting the rest of the evaluation. However, rule P.1 is restricted and operates only on begin expressions. To mitigate this restriction, rules P.3 and P.5 reshape expressions that contain unreachable into begin expressions. Since a compiler should be able to perform transformations in any context, the Ctx.P rule lifts all of the  $R_p$  rules to arbitrary contexts. Also, all of the transformations in  $R_p$  are legal in either direction, so CtxSym.P adds in the symmetric variants.

Figure 6 presents  $\rightarrow_u$ . Rules U.1 and U.2 in figure 6 define the essential notions of reduction and they capture the undefined behavior of unreachable, matching the behavior of Racket’s `unsafe-assert-unreachable` and Rust’s `unreachable_unchecked`. They specify how a compiler can eliminate unreachable branches of conditionals. In other words, they are the formal counterparts of the essential transformation steps that the Racket compiler performs in the examples from section 2.

The  $R_u$  axioms are sound in any context but, unlike  $\rightarrow_p$ , they are not sound in reverse. The issue is that the  $R_p$  notions of reductions eliminate immaterial pieces of an expression. However, a symmetric  $\rightarrow_u$  reduction would inject arbitrary expressions back, possibly affecting the meaning

of the expression. For instance, the compiler is only justified to transform (if  $e$  is unreachable) to (begin  $e$ ) because it assumes that this unreachable is indeed unreachable. Reversing the reduction, though, introduces an unreachable for which the compiler knows nothing about. This asymmetry is the source of the main challenge for proving the correctness of the compile-time reductions, and we revisit the issue in section 4.

#### 4 THE CORRECTNESS OF THE COMPILE-TIME REDUCTIONS

The simplicity of the compile-time reductions of the unreachable calculus comes with a price: establishing their correctness is challenging. The root of the challenge is that unreachable is a kind of undefined behavior, and this affects radically what the correctness of the reductions means. In general, the ultimate correctness criterion of any program transformation is that it preserves the meaning of programs. However, transformations that take advantage of undefined behavior, such as those captured by  $\rightarrow_c$ , are supposed to preserve the meaning only of programs that do not exhibit undefined behavior; the rest are outside the universe of programs a compiler should handle. In other words, in a program that exhibits undefined behavior,  $\rightarrow_c$  has the liberty to alter the program's behavior. It is this liberty that impedes a compositional proof of correctness for the reductions. Specifically, a compositional proof requires reasoning about whether a compiler preserves the meaning of a piece of a program in isolation from the rest of the program, but undefined behavior can only be determined for a whole program.

Fortunately, unreachable is a rather “well-behaved” undefined behavior. In particular, the small set of intuitive compile-time reductions from section 3 have the property that they preserve the meaning of any expression in any context under certain conditions. In turn, these conditions can be described with a nonstandard, novel logical relation that hides the complexity of the proof of correctness, keeping the syntax and the reductions of the calculus simple.

To get a sense of the correctness theorem we are aiming for, consider two complete programs  $e$  and  $e'$  such that  $e \rightarrow_c e'$ , i.e., where  $e'$  is a program that the compiler is allowed to transform  $e$  into. Accordingly, if the evaluation of  $e$  does not exhibit undefined behavior, we wish to ensure that  $e'$  does not exhibit undefined behavior either and that  $e'$  and  $e$  either both diverge or both terminate with the same result.

Translating this informal description into a formal statement requires making the notion of undefined behavior precise, using the standard reduction relation:

*Definition 4.1 (Undefined Behavior).* The behavior of a closed expression  $e$  is *undefined*, written as  $\text{UNDEF}(e)$ , if  $e \rightarrow_s^* \text{unreachable}$ .

which leads to the formal definition of compiler correctness:

**PROPOSITION 4.2.** For all closed expressions  $e, e'$  such that  $e \rightarrow_c^* e'$ , if  $\neg \text{UNDEF}(e)$  then

- $\neg \text{UNDEF}(e')$ ,
- $\forall c. (e \rightarrow_s^* c \iff e' \rightarrow_s^* c)$ ,
- $(\exists e_1. e \rightarrow_s^* \lambda x. e_1) \iff (\exists e'_1. e' \rightarrow_s^* \lambda x. e'_1)$ , and
- $\forall k. (e \rightarrow_s^* \text{error}_k \iff e' \rightarrow_s^* \text{error}_k)$ .

*Remark.* Because both  $e$  and  $e'$  do not terminate with unreachable, the conclusion of proposition 4.2 implies that non-termination is preserved by  $\rightarrow_c$ ;  $e'$  diverges if and only if  $e$  diverges.

The standard, logical-relations based approach to proving proposition 4.2 is to define a step-indexed syntax-based binary relation that uses the standard reduction to reduce expressions to values. This logical relation captures a notion of logical approximation such that two expressions that approximate each other are contextually equivalent. Soundness of the logical relation with



$$\begin{aligned}
f_{\text{src}} &:= \lambda p x. (994 + (\text{if } (p \ x) \ \text{unreachable } x)) \\
f_{\text{opt}} &:= \lambda p x. (994 + (\text{begin } (p \ x) \ x)) \\
f_{\text{src}} (\lambda y. \text{false}) n &\rightarrow_s^* 994 + n & f_{\text{src}} (\lambda y. \text{true}) n &\rightarrow_s^* \text{unreachable} \\
f_{\text{opt}} (\lambda y. \text{false}) n &\rightarrow_s^* 994 + n & f_{\text{opt}} (\lambda y. \text{true}) n &\rightarrow_s^* 994 + n
\end{aligned}$$

Fig. 7. Examples of Semantics-Altering Transformations Due to  $\rightarrow_u$  Reductions.

respect to contextual equivalence is established by showing its Fundamental Property, i.e., any expression  $e$ , if  $\neg \text{UNDEF}(e)$ , then  $e$  is related to itself.

Once that is done, one would attempt to prove the compile-time reductions correct by showing that the left-hand side and the right-hand side of each reduction rule logically approximate each other or, more precisely, if  $\neg \text{UNDEF}(e)$  and  $e \rightarrow_c e'$ , then  $e$  is related to  $e'$  and  $e'$  is related to  $e$ .

Unfortunately, this approach does not work because of the way  $\text{UNDEF}$  is defined. Consider the situation where we know that some application expression  $e_s e'_s$  that reduces via  $\rightarrow_c$  to  $e_c e'_c$  and we wish to show that  $e_c e'_c$  is related to  $e_s e'_s$ . In this case, we do not benefit from the  $\neg \text{UNDEF}(e_s e'_s)$  assumption. In essence,  $\neg \text{UNDEF}(e_s e'_s)$  does not translate to facts about the pieces of  $e_s e'_s$  that we can map to the the pieces of  $e_c e'_c$  to complete the proof inductively.

Functions  $f_{\text{src}}$ ,  $f_{\text{opt}}$  in Figure 7 demonstrate the issue. Consider a pair of programs  $(e, e') := (f_{\text{src}} (\lambda y. b) n, f_{\text{opt}} (\lambda y. b) n)$ . Given that  $e' \rightarrow_s^* 994 + n$  for all  $n$  and  $b$ , we would like to be able to use the logical relation to deduce that  $e \rightarrow_s^* 994 + n$  based entirely on the fact that  $(f_{\text{src}}, f_{\text{opt}})$  are related. But as figure 7 demonstrates,  $e$  exhibits undefined behavior when  $b \equiv \text{false}$ . In fact, the conditions under which  $e$  exhibits undefined behavior become involved if we consider an arbitrary argument  $p$  that can exhibit undefined behavior:

$$\begin{aligned}
f_{\text{src}} (\lambda y. (\text{if } (y = 0) \ \text{unreachable } b)) n &\rightarrow_s^* 994 + n &\iff &b \equiv \text{false} \wedge n \neq 0 \\
f_{\text{opt}} (\lambda y. (\text{begin } (y = 0) \ b)) n &\rightarrow_s^* 994 + n
\end{aligned}$$

Put differently, the assumption  $\neg \text{UNDEF}(e)$  has turned into an arbitrary constraint on the arguments of the two functions, which is unclear how to incorporate in a logical relation.

The literature on logical relations suggests a way around the problem. One can approximate a global property, such as  $\text{UNDEF}$ , with an inductively-defined approximate predicate. For example, RustBelt (Jung et al. 2018) takes advantage of Rust's type system and ensures the absence of undefined behavior based on a semantic type judgment. In other words, such predicates aim to break the whole-program property into compositional facts about the structural pieces of an expression. Unfortunately, such an approach does not work here in a straightforward manner because  $\text{UNDEF}$  simply is not compositional.

Instead of attempting to come up with some conservative inductive substitute for  $\text{UNDEF}$ , we follow a different path that aims to keep the calculus as close to the way compiler writers use the full-blown, non-compositional definition of undefined behavior when reasoning about compiler transformations. We define forward- and backward-approximation logical relations that, when an expression does not exhibit undefined behavior, relate it with its transformed version after a sequence of  $\rightarrow_c$  reductions, and vice versa. In fact, we define four such relations: two for  $\rightarrow_u$  and two for  $\rightarrow_p$  reductions. Uncharacteristically, the forward and backward approximations for  $\rightarrow_u$  reductions do not mirror each other. This complicates showing their soundness but achieves the goal of hiding all complexity away from the calculus.

Before delving into the details of the logical relations, we first introduce well-formedness judgments for managing free variables since the fundamental properties for the logical relations assume

$$\begin{array}{c}
\frac{x \in \Delta}{\Delta \Vdash x} \text{W.VAR} \quad \boxed{\Delta \Vdash e} \quad \frac{\Delta' \supseteq \Delta}{\Delta' \Vdash [] : \Delta} \text{C.ID} \quad \frac{\Delta', x \Vdash C : \Delta}{\Delta' \Vdash \lambda x. C : \Delta} \text{C.LAMBDA} \quad \boxed{\Delta' \Vdash C : \Delta} \\
\frac{\Delta, x \Vdash e}{\Delta \Vdash \lambda x. e} \text{W.LAMBDA} \quad \frac{\Delta' \Vdash C : \Delta \quad \Delta' \Vdash e_t \quad \Delta' \Vdash e_f}{\Delta' \Vdash (\text{if } C e_t e_f) : \Delta} \text{C.IFC}
\end{array}$$

Fig. 8. Well-formed Expressions and Contexts (Selected Rules)

open expressions. Figure 8 presents a few selected inference rules. The complete list of wellformedness rules can be found on Appendix B. In the figure,  $\Delta$  and  $\Delta'$  denote sets of variables. The judgment  $\Delta \Vdash e$  holds if the set of free variables in the expression  $e$  is a subset of  $\Delta$ . The judgment  $\Delta' \Vdash C : \Delta$  asserts that the context  $C$  maps an expression that refers to variables in  $\Delta$  into an expression that refers to variables in  $\Delta'$ . Put differently, if  $\Delta \Vdash e$  and  $\Delta' \Vdash C : \Delta$  then  $\Delta' \Vdash C[e]$ .

The remainder of this section establishes the correctness of the  $\rightarrow_c$  reductions. The first two subsections describe the forward and backward approximation logical relations for  $\rightarrow_u$  reductions and their soundness. Then, the final subsection discusses the correctness of the  $\rightarrow_p$  reductions to complete the proof of correctness of our compile-time reductions.

#### 4.1 Forward Approximation of $\rightarrow_u$ Reductions

As a first step to prove the correctness of  $\rightarrow_u$  reductions, we design the binary Forward-Approximation Logical Relation. In detail, for all  $i \geq 0$ , we define the step-indexed logical relation for values and *closed* expressions as  $\mathcal{V}_i^{\rightarrow u}$  and  $\mathcal{E}_i^{\rightarrow u}$ , respectively. In the definition, the notation  $e \rightarrow_s^j e'$  means that  $e$  reduces to  $e'$  using exactly  $j$  steps under the standard reduction of the calculus.

*Definition 4.3 (Forward-Approximation Logical Relation).*

$$\begin{aligned}
\mathcal{V}_i^{\rightarrow u} &= \{(\lambda x. e_1, \lambda x. e_2) \mid \forall j < i. \forall v_1 v_2. (v_1, v_2) \in \mathcal{V}_j^{\rightarrow u}, (e_1 [v_1/x], e_2 [v_2/x]) \in \mathcal{E}_j^{\rightarrow u}\} \\
&\quad \cup \{(c, c)\} \\
\mathcal{E}_i^{\rightarrow u} &= \{(e_1, e_2) \mid \forall j < i. \forall a_1. (\neg \text{UNDEF}(e_1)) \wedge e_1 \rightarrow_s^j a_1 \Rightarrow \\
&\quad \forall e'_2. e_2 \rightarrow_u^* e'_2 \Rightarrow \\
&\quad \exists a_2. e'_2 \rightarrow_s^* a_2 \wedge ((a_1, a_2) \in \mathcal{V}_{i-j}^{\rightarrow u} \vee \exists k. a_1 \equiv a_2 \equiv \text{error}_k)\}
\end{aligned}$$

*Remark.* The definition of  $\mathcal{V}_i^{\rightarrow u}$  uses  $\mathcal{V}_j^{\rightarrow u}$  and  $\mathcal{E}_j^{\rightarrow u}$  for  $j$  that is strictly less than  $i$ , and that the definition of  $\mathcal{E}_i^{\rightarrow u}$  uses  $\mathcal{V}_{i-j}^{\rightarrow u}$  for  $0 \leq j < i$ . Thus the mutually-referential relations are well-founded.

The value relation  $\mathcal{V}_i^{\rightarrow u}$  is standard. Two values  $(v_1, v_2)$  are related by  $\mathcal{V}_i^{\rightarrow u}$  at step  $i \geq 0$  if  $v_1$  and  $v_2$  are the same constant  $c$ , or if they are both lambdas and, for all  $j < i$ , applying arguments that are related at  $j$  steps produces expressions related at  $j$  steps.

The expression relation  $\mathcal{E}_i^{\rightarrow u}$ , in contrast, is not standard. It is crafted to resemble one of the directions of proposition 4.2, but specialized to  $\rightarrow_u$  reductions. Specifically, under the assumption that  $\neg \text{UNDEF}(e_1)$  and  $e_1$  evaluates to an answer after  $j < i$  standard reduction steps,  $e_1$  is related to  $e_2$  at step index  $i$  if all expressions  $e'_2$  that are results of simplifying  $e_2$  with  $\rightarrow_u$  reductions evaluate to answers that are related to  $e_1$ 's answer. Two answers are related if they are the same error, or if they are related values at step index  $i - j$ .

In other words, the expression approximation aims to establish directly that after some  $\rightarrow_u$  reductions the resulting expression approximates the meaning of the initial one. For that reason, the expression approximation has two built-in assumptions that are not found in standard logical relations:  $\neg \text{UNDEF}(e_1)$  and  $e_2 \rightarrow_u^* e'_2$ . As discussed above, in a standard logical relation the first would be an assumption for its Fundamental Property, while the second would be an assumption

of a separate theorem that uses the soundness of the logical relation to prove that  $\rightarrow_u$  reductions are correct.

Equipped with these relations, we are ready to prove the Fundamental Property of  $\mathcal{E}^{\rightarrow u}$ . As usual, it states that any open expression  $e$  is related to itself:

LEMMA 4.4 (FUNDAMENTAL PROPERTY OF  $\mathcal{E}^{\rightarrow u}$ ). *For all  $e, i \geq 0, \Delta$  and  $\gamma$ , if  $\Delta \Vdash e$  and  $\gamma \in \mathcal{G}_i^{\rightarrow u}[\Delta]$  then  $(\gamma_1(e), \gamma_2(e)) \in \mathcal{E}_i^{\rightarrow u}$ .*

*Remark.*  $\mathcal{G}_i^{\rightarrow u}$  is the standard relation on pairs of substitutions that map the same identifier to values related by  $\mathcal{V}_i^{\rightarrow u}$ . The complete definition can be found on page 4 of Appendix E.

PROOF SKETCH. We prove the Fundamental Property by induction on  $e$ . In most cases, the transformation  $\gamma_2(e) \rightarrow_u^* e'_2$  does not change the outermost shape of  $\gamma_2(e)$  and hence the proofs are straightforward. When  $e ::= (\text{if } e_c \text{ unreachable } e_f)$ , the branch-elimination transformation may reduce  $\gamma_2((\text{if } e_c \text{ unreachable } e_f))$  to  $(\text{begin } \gamma_2(e_c) \gamma_2(e_f))$ .<sup>2</sup> In this case, we need to prove that the latter reduces to a related answer based on the assumption that  $\gamma_1((\text{if } e_c \text{ unreachable } e_f)) \rightarrow_s^j a_1$  and  $a_1 \not\equiv \text{unreachable}$ . As it turns out, this variation poses no issue to the proof because the sub-expressions reduce in a related manner by induction. The complete proof can be found on page 6 of Appendix E.  $\square$

A consequence of the Fundamental Property is the forward direction of proposition 4.2 specialized to  $\rightarrow_u$ : when  $e$  does not exhibit undefined behavior, for any transformation  $e \rightarrow_u^* e'$ , if  $e$  terminates then  $e'$  terminates with a related answer.

COROLLARY 4.5. *Assume  $\Delta \Vdash e$  and  $e \rightarrow_u^* e'$ . For all  $C, a$ , if  $\Vdash C : \Delta, \neg \text{UNDEF}(C[e])$  and  $C[e] \rightarrow_s^* a$  then there exists  $a'$  and  $j \geq 0$  such that  $C[e'] \rightarrow_s^* a'$  and either  $(a, a') \in \mathcal{V}_j^{\rightarrow u}$  or  $a \equiv a' \equiv \text{error}_k$ .*

PROOF SKETCH. Assume  $C[e] \rightarrow_s^i a$ . Because  $\rightarrow_u$  allows us to apply the transformation in any context, composing  $C$  with each of the expressions in  $e \rightarrow_u^* e'$  yields  $C[e] \rightarrow_u^* C[e']$ . Now, the Fundamental Property of  $\mathcal{E}^{\rightarrow u}$  gives  $(C[e], C[e']) \in \mathcal{E}_{i+1}^{\rightarrow u}$ . By  $\neg \text{UNDEF}(C[e])$ ,  $C[e] \rightarrow_s^i a$  and  $C[e] \rightarrow_u^* C[e']$ , we conclude that there exists  $a_2$  such that  $C[e'] \rightarrow_s^* a_2$  and either  $(a, a_2) \in \mathcal{V}_1^{\rightarrow u}$  or  $a \equiv a_2 \equiv \text{error}_k$ . The complete proof can be found on page 4 of Appendix E.  $\square$

## 4.2 Backward Approximation of $\rightarrow_u$ Reductions

Having established the forward direction of proposition 4.2 for  $\rightarrow_u$ , we turn to the backward one. That is, we would like to show that the behavior of the original expression approximates the behavior of the transformed expression, assuming that the original expression does not exhibit undefined behavior: if  $e \rightarrow_u^* e'$  and  $\neg \text{UNDEF}(e)$  then

- $\neg \text{UNDEF}(e')$ ,
- $\forall c. (e \rightarrow_s^* c \iff e' \rightarrow_s^* c)$ ,
- $(\exists e_1. e \rightarrow_s^* \lambda x. e_1) \iff (\exists e'_1. e' \rightarrow_s^* \lambda x. e'_1)$ , and
- $\forall k. (e \rightarrow_s^* \text{error}_k \iff e' \rightarrow_s^* \text{error}_k)$ .

However, as discussed at the beginning of this section, the  $\neg \text{UNDEF}(e)$  assumption complicates designing a backward-approximation logical relation. In an ideal world, we would like to proceed by induction on  $e'$ , as our assumption tells us a lot about how it evaluates. Unfortunately, our assumption also includes  $\neg \text{UNDEF}(e)$ , which is not helpful when we are working by induction on  $e'$ .

<sup>2</sup>The actual proof needs to generalize  $\gamma_2(e_c)$  and  $\gamma_2(e_f)$  further. See proposition 4.8.

As it turns out, the treatment of unreachable by the standard reduction of our calculus as an error offers a way forward. While  $\neg\text{UNDEF}(e)$  is not compositional,  $\text{UNDEF}(e)$  is. In detail, if we know whether the sub-expressions of  $\text{UNDEF}(e)$  evaluate to unreachable, we can decide whether  $e \rightarrow_s^*$  unreachable based on the structure of  $e$ . Accordingly, while working by induction on  $e'$  in the proof, it is easier to establish an additional proof goal  $\text{UNDEF}(e)$  than it is to discharge the premise that the sub-expressions of  $e$  do not evaluate to unreachable (when applying the inductive hypothesis). Therefore, we reshape the statement of backward approximation: we omit conjunct  $\neg\text{UNDEF}(e)$  from the premise and include  $\text{UNDEF}(e)$  as another disjunct in the conclusion of backward approximation. This gives us an equivalent proposition, but that is conducive to proof by induction.

To prove the reshaped property, we construct a logical relation that guarantees the original expression produces a related answer to that of the transformed expression, but also permits the original expression to exhibit undefined behavior, just as we did for the forward logical relation.

*Definition 4.6 (Backward-Approximation Logical Relation).*

$$\begin{aligned} \mathcal{V}_i^{u\leftarrow} &= \{(\lambda x.e_1, \lambda x.e_2) \mid \forall j < i. \forall v_1 v_2. (v_1, v_2) \in \mathcal{V}_j^{u\leftarrow} \Rightarrow (e_1 [v_1/x], e_2 [v_2/x]) \in \mathcal{E}_j^{u\leftarrow}\} \\ &\quad \cup \{(c, c)\} \\ \mathcal{E}_i^{u\leftarrow} &= \{(e_1, e_2) \mid \forall j < i. \forall e'_2, a_2. e_2 \rightarrow_u^* e'_2 \wedge e'_2 \rightarrow_s^j a_2 \Rightarrow \\ &\quad \text{UNDEF}(e_1) \vee \\ &\quad (\exists a_1. e_1 \rightarrow_s^* a_1 \wedge ((a_1, a_2) \in \mathcal{V}_{i-j}^{u\leftarrow} \vee \exists k. a_1 \equiv a_2 \equiv \text{error}_k))\} \end{aligned}$$

Same as for the forward-approximation logical relation, the value relation  $\mathcal{V}_i^{u\leftarrow}$  is standard, but the expression relation  $\mathcal{E}_i^{u\leftarrow}$  is not. For any pair of related expressions  $(e_1, e_2)$ , the antecedent of  $\mathcal{E}_i^{u\leftarrow}$  is that  $e_2 \rightarrow_u^* e'_2$  and  $e'_2$  terminates with the answer  $a_2$  after  $j$  standard reduction steps. The consequent then asserts that either  $e_1$  exhibits undefined behavior, or  $e_1$  also terminates with a related answer  $a_1$ . In particular,  $\mathcal{E}_i^{u\leftarrow}$  relates  $a_1$  and  $a_2$  if they are related *values* or the same error. Consequently, if  $a_2 \equiv \text{unreachable}$ , i.e.  $e'_2$  triggers undefined behavior,  $e_1$  must end up with undefined behavior as unreachable is not related to any other answers.

As with  $\mathcal{E}^{\rightarrow u}$ , we prove the Fundamental Property for  $\mathcal{E}^{u\leftarrow}$ .

**LEMMA 4.7 (FUNDAMENTAL PROPERTY FOR  $\mathcal{E}^{u\leftarrow}$ ).** *For all  $e, i \geq 0, \Delta$  and  $\gamma$ , if  $\Delta \Vdash e$  and  $\gamma \in \mathcal{G}_i^{u\leftarrow}[\Delta]$  then  $(\gamma_1(e), \gamma_2(e)) \in \mathcal{E}_i^{u\leftarrow}$ .*

*Remark.* Similar to the Fundamental Property for the forward logical approximation,  $\mathcal{G}_i^{u\leftarrow}[\Delta]$  is the set of closing substitutions mapping  $x \in \Delta$  to a pair of values related by  $\mathcal{V}_i^{u\leftarrow}$ .

**PROOF SKETCH.** By induction on  $e$ . Similar to the Fundamental Property of  $\mathcal{E}^{\rightarrow u}$ , we discuss the case of conditional expressions. The proofs for the other constructs follow from routine case analysis and the inductive hypothesis. Let  $e := (\text{if } e_c e_t e_f)$ ,  $i \geq 0$  and  $\gamma \in \mathcal{G}_i^{u\leftarrow}[\Delta]$  be given. We need to prove that

$$\begin{aligned} \forall j < i. \forall e'_2 a_2. (\text{if } \gamma_2(e_c) \gamma_2(e_t) \gamma_2(e_f)) \rightarrow_u^* e'_2 \wedge e'_2 \rightarrow_s^j a_2 \Rightarrow \\ \text{UNDEF}(\text{if } \gamma_1(e_c) \gamma_1(e_t) \gamma_1(e_f)) \vee \\ (\exists a_1. (\text{if } \gamma_1(e_c) \gamma_1(e_t) \gamma_1(e_f)) \rightarrow_s^* a_1 \wedge ((a_1, a_2) \in \mathcal{V}_{i-j}^{u\leftarrow} \vee \exists k. a_1 \equiv a_2 \equiv \text{error}_k)) \end{aligned}$$

By proposition 4.8,  $(\text{if } \gamma_2(e_c) \gamma_2(e_t) \gamma_2(e_f)) \rightarrow_u^* e'_2$  can be decomposed into three reduction sequences  $\gamma_2(e_c) \rightarrow_u^* e'_c$ ,  $\gamma_2(e_t) \rightarrow_u^* e'_t$  and  $\gamma_2(e_f) \rightarrow_u^* e'_f$  such that either (i)  $e'_2 \equiv (\text{if } e'_c e'_t e'_f)$ , (ii)  $e'_2 \equiv (\text{begin } e'_c e'_t)$  and  $e'_f \equiv \text{unreachable}$ , or (iii)  $e'_2 \equiv (\text{begin } e'_c e'_f)$  and  $e'_t \equiv \text{unreachable}$ . Among all situations, we focus on case (iii) since it contains an application of  $\rightarrow_u$  to the whole expression.

Now, if  $a_2$  equals some value  $v'_f$ , the evaluation  $(\text{begin } e'_c e'_f) \rightarrow_s^j a_2$  must have the pattern

$$(\text{begin } e'_c e'_f) \rightarrow_s^{j_c} (\text{begin } v'_c e'_f) \rightarrow_s e'_f \rightarrow_s^{j_f} v'_f.$$

Therefore  $e'_c \rightarrow_s^{j_c} v'_c$  and  $e'_f \rightarrow_s^{j_f} v'_f$  for some steps  $j'_c$  and  $j'_f$ . Together with the transformations  $\gamma_2(e_c) \rightarrow_u^* e'_c$  and  $\gamma_2(e_f) \rightarrow_u^* e'_f$ , the induction hypothesis yields

$$\begin{aligned} & \text{UNDEF}(\gamma_1(e_c)) \vee (\exists a_c. \gamma_1(e_c) \rightarrow_s^* a_c \wedge ((a_c, v'_c) \in \mathcal{V}_{i-j_c}^{u \leftarrow} \vee \exists k. a_c \equiv v'_c \equiv \text{error}_k)) \\ & \text{UNDEF}(\gamma_1(e_f)) \vee (\exists a_f. \gamma_1(e_f) \rightarrow_s^* a_f \wedge ((a_f, v'_f) \in \mathcal{V}_{i-j_c-1-j_f}^{u \leftarrow} \vee \exists k. a_f \equiv v'_f \equiv \text{error}_k)) \end{aligned} \quad (\star)$$

Because  $v'_c$  ( $v'_f$ ) is a value, the  $\text{error}_k$  case in  $(\star)$  cannot happen. The answer  $a_c$  ( $a_f$ ) accordingly is related to  $v'_c$  ( $v'_f$ ). As a result, if  $\text{UNDEF}(\gamma_1(e_c))$  or  $\text{UNDEF}(\gamma_1(e_f))$ , i.e. a subexpression triggers undefined behavior, the treatment of unreachable by the standard reduction guarantees that  $\text{UNDEF}(\text{if } \gamma_1(e_c) \gamma_1(e_t) \gamma_1(e_f))$ . Otherwise, we have  $\gamma_1(e_c) \rightarrow_s^* a_c \wedge (a_c, v'_c) \in \mathcal{V}_{i-j_c}^{u \leftarrow}$  and  $\gamma_1(e_f) \rightarrow_s^* a_f \wedge (a_f, v'_f) \in \mathcal{V}_{i-j_c-1-j_f}^{u \leftarrow}$ . The evaluation of  $(\text{if } \gamma_1(e_c) \gamma_1(e_t) \gamma_1(e_f))$  thus depends on whether  $a_c$  is false or not.

When  $a_c$  is false,  $(\text{if } \gamma_1(e_c) \gamma_1(e_t) \gamma_1(e_f))$  reduces  $\gamma_1(e_f)$ . Hence the fact  $(a_f, v'_f) \in \mathcal{V}_{i-j_c-1-j_f}^{u \leftarrow}$  entails our desired conclusion. When  $a_c$  is truthy,  $(\text{if } \gamma_1(e_c) \gamma_1(e_t) \gamma_1(e_f))$  reduces to  $\gamma_1(e_t)$ . Since  $\gamma_2(e_t) \rightarrow_u^* e'_t$  and  $e'_t \equiv \text{unreachable}$  in case (iii),  $e_t$  itself must be unreachable. Hence, the full expression  $(\text{if } \gamma_1(e_c) \gamma_1(e_t) \gamma_1(e_f))$  terminates with unreachable. Nevertheless, exhibiting undefined behavior is precisely one of the expected behaviors.

At this point, we have covered the sub-case  $a_2 \equiv v'_f$  of case (iii). The sub-case where  $a_2$  is unreachable is proved with the same strategy. The complete proof can be found on page 5 of Appendix F.  $\square$

The proof of the Fundamental Property for  $\mathcal{E}^{u \leftarrow}$  relies on the following property of sequences of  $\rightarrow_u$  reductions:

**PROPOSITION 4.8.** *If  $(\text{if } e_c e_t e_f) \rightarrow_u^* e'$ , there exists three transformation sequences  $e_c \rightarrow_u^* e'_c$ ,  $e_t \rightarrow_u^* e'_t$  and  $e_f \rightarrow_u^* e'_f$  such that one of the following holds:*

- $e' \equiv (\text{if } e'_c e'_t e'_f)$
- $e' \equiv (\text{begin } e'_c e'_t)$  and  $e'_f \equiv \text{unreachable}$ , or
- $e' \equiv (\text{begin } e'_c e'_f)$  and  $e'_t \equiv \text{unreachable}$

**PROOF SKETCH.** By induction on  $(\text{if } e_c e_t e_f) \rightarrow_u^* e'$ . The complete proof can be found on page 12 of Appendix C.  $\square$

Analogously to the Fundamental Property for the forward-approximation relation, a corollary of the Fundamental Property for  $\mathcal{E}^{u \leftarrow}$  is the backward direction of proposition 4.2 specialized to  $\rightarrow_u$ : either the transformed program and the original program evaluate to related answers, or the original program exhibits undefined behavior:

**COROLLARY 4.9.** *Assume that  $\Delta \Vdash e$  and  $e \rightarrow_u^* e'$ . For all  $C$  and  $a'$ , if  $\Vdash C : \Delta$  and  $C[e'] \rightarrow_s^* a'$  then either  $\text{UNDEF}(C[e])$  or there exists  $a$  and  $j \geq 0$  such that  $C[e] \rightarrow_s^* a$  and either  $(a, a') \in \mathcal{V}_j^{\rightarrow u}$  or  $a \equiv a' \equiv \text{error}_k$ .*

*Remark.* In corollary 4.9,  $a'$  can be unreachable in which case the conclusion is  $\text{UNDEF}(C[e])$ .

PROOF SKETCH. Assume  $C[e'] \rightarrow_s^i a'$ . Similar to corollary 4.5, we compose  $C$  with each expression in  $e \rightarrow_u^* e'$  to obtain  $C[e] \rightarrow_u^* C[e']$ . By the Fundamental Property for  $\mathcal{E}^{u\leftarrow}$ ,  $(C[e], C[e']) \in \mathcal{E}_{i+1}^{u\leftarrow}$ . The fact that  $C[e] \rightarrow_u^* C[e']$  and  $C[e'] \rightarrow_s^i a'$  yields  $\text{UNDEF}(C[e])$  or that there exists  $a_1$  such that  $C[e] \rightarrow_s^* a_1$  and either  $(a_1, a') \in \mathcal{V}_1^{u\leftarrow}$  or  $a_1 \equiv a' \equiv \text{error}_k$ . The complete proof can be found on page 2 of Appendix F.  $\square$

### 4.3 Completing the Proof of Correctness of the Compile-Time Reductions

The fundamental properties of the two above logical relations entail only one piece of the correctness of our compile-time reduction relation. After all,  $\rightarrow_c$  includes  $\rightarrow_p$  as well as  $\rightarrow_u$ .

Fortunately, and unlike the challenging  $\rightarrow_u$  reductions,  $\rightarrow_p$  reductions preserve the meaning of expressions in all contexts without any conditions. Hence, the proof of proposition 4.2 specialized to  $\rightarrow_p$ , i.e. the correctness of the  $\rightarrow_p$  reductions, can be established the standard way we discuss at the beginning of this section – with a standard logical relation. Indeed, the correctness of the  $\rightarrow_p$  reductions is a simple corollary of lemma 5.3, which we discuss in Section 5 that introduces extensions of the compile-time reduction with useful, semantics-preserving but unrelated-to-unreachable transformations. To avoid repetition, we omit further discussion herein. The interested reader can also find the the full formal details in page 28 of Appendix G and Appendix H.

With the correctness of  $\rightarrow_u$  and  $\rightarrow_p$  in hand, we proceed to prove the correctness of  $\rightarrow_c$ . In fact, we establish a generalized compiler correctness theorem that adapts proposition 4.2 to open expressions, formalizing the intuitive idea that compiler transformations preserve the semantics of program pieces. In keep with the nature of undefined behavior, the theorem only holds for contexts  $C$  that close an expression  $e$  so that  $\neg\text{UNDEF}(C[e])$ :

THEOREM 4.10 (GENERALIZED COMPILER CORRECTNESS).

Assume that  $\Delta \Vdash e$  and  $e \rightarrow_c^* e'$ . For all  $C$  such that  $\Vdash C : \Delta$ , if  $\neg\text{UNDEF}(C[e])$  we have

- $\neg\text{UNDEF}(C[e'])$ ,
- $\forall c. C[e] \rightarrow_s^* c \iff C[e'] \rightarrow_s^* c$ ,
- $(\exists e_1. C[e] \rightarrow_s^* \lambda x. e_1) \iff (\exists e'_1. C[e'] \rightarrow_s^* \lambda x. e'_1)$ , and
- $\forall k. C[e] \rightarrow_s^* \text{error}_k \iff C[e'] \rightarrow_s^* \text{error}_k$

PROOF SKETCH. By induction on  $e \rightarrow_c^* e'$  and application of corollary 4.5 and corollary 4.9 for each  $\rightarrow_u$  reduction. For  $\rightarrow_p$  reductions, we apply lemma 5.3. The complete proof can be found on Appendix J.  $\square$

## 5 EXTENDING THE CALCULUS WITH ADDITIONAL RULES

The  $\rightarrow_u$  and  $\rightarrow_p$  reductions capture the essence of unreachable but they are not sufficient to describe realistic compile-time transformations. Production compilers combine common transformations such as Common Subexpression Elimination, Loop Unrolling, and Strength Reduction with the unreachable-related transformations; the transformations work in tandem, as one transformation may open up additional optimization opportunities for another. For this reason, we add extra rules to the unreachable calculus that enhance its power to capture realistic program transformations. In fact, any extra rule is compatible with the calculus under one requirement: the new rule must be sound with respect to contextual equivalence as induced by the standard reduction of the calculus.

As an example of extra compile-time rules, rules M.1 to M.5 in Figure 9 make available at compile-time the standard notions of reduction of the calculus from Figure 3. In essence, they allow the compiler to partially “compute” forward and backward in any sub-expression of a program:

- M.1 corresponds to the reverse notion of reduction S.2. It enables a compiler to “reverse the evaluation” of conditional expressions whose test is known to be not false. The else-branch

$$\begin{array}{c}
\frac{e \rightarrow_m e'}{e \rightarrow_c e'} \quad \frac{\Delta \Vdash e_t \quad \Delta \Vdash e_f \quad v \neq \text{false}}{\Delta \vdash e_t R_m(\text{if } v e_t e_f)} \text{ M.1} \quad \frac{\boxed{e \rightarrow_c e} \quad \boxed{\Delta \vdash e R_m e} \quad \boxed{e \rightarrow_m e}}{\Delta \Vdash e_f \quad \Delta \Vdash e_t} \text{ M.2} \\
\frac{\Delta, x \Vdash e \quad \Delta \Vdash e' \quad \text{SAFE}(e')}{\Delta \vdash e[e'/x] R_m(\lambda x. e) e'} \text{ M.3} \\
\frac{\Delta \Vdash e_1 \quad \Delta \Vdash e_2 \quad \text{SAFE}(e_1)}{\Delta \vdash (\text{begin } e_1 e_2) R_m e_2} \text{ M.4} \quad \frac{\delta(\text{op}, v_1, v_2) = c}{\Delta \vdash c R_m(\text{op } v_1 v_2)} \text{ M.5} \\
\frac{\cdot \Vdash C : \Delta \quad \Delta \vdash e R_m e'}{C[e] \rightarrow_m C[e']} \text{ CTX.M} \quad \frac{\cdot \Vdash C : \Delta \quad \Delta \vdash e' R_m e}{C[e] \rightarrow_m C[e']} \text{ CTXSYM.M}
\end{array}$$

Fig. 9. Compile-Time Transformations Based on Standard Reduction

of the produced conditional expression can be any arbitrary expression as long as it doesn't introduce free variables not accounted for by  $\Delta$  – even expressions that contain unreachable.

- M.2 is the reverse of notion of reduction S.1, and similarly to M.1 produces conditional expression that in this case have a test that is equal to false.
- M.3 is a generalized reversed beta reduction. Specifically, it allows any *safe* sub-expression to be lifted out of an expression, which, in the compiler realm, is useful for modeling transformations such as Common Subexpression Elimination.
- M.4 corresponds to the notion of reduction S.4. It permits the compiler to drop all expressions in a sequence of expressions except the last one as long as these dropped expressions evaluate to a value. This rule together with rules U.1 and U.2 plays an important rule for the simplification of the examples in Section 2.
- M.5 is the reverse of notion of reduction S.5, which allows for the backward “evaluation” of arithmetic expressions.
- CTX.M and CTXSYM.M define the symmetric and compatible closure (over contexts) of the above rules. Hence, they allow compilers to use all these rules forward and backward, and in any part of a program.

In addition to the forward and backward compile-time partial evaluation of expressions, compilers come with a large number of transformation rules that move the sub-expressions of a program. Figure 10 contains a collection of such rules.

- M.6 and M.7 enable the compiler to move computation from the context of a sequence and a conditional expression respectively to tail position. The  $E^+$  represents a generalized version of evaluation context which admits variables.
- M.8 allows the compiler to optimize conditional expressions whose test is a variable (say  $x$ ). If the else-branch of the expression is ever evaluated, then it must be the case that  $x$  is false. Note the same is not correct for the then-branch and true due to the truthiness of the language.
- M.9 permits the compiler to collapse certain conditional expressions whose test is a trivial conditional statement as well. This is helpful for modeling optimizations for languages with conditional select statements such as the LLVM IR.

$$\begin{array}{c}
\boxed{\Delta \Vdash C : \Delta} \quad \boxed{\Delta \vdash e R_m e} \quad \boxed{\Delta \Vdash e} \\
v^+ ::= x \mid v \\
E^+ ::= [] \mid op E^+ e \mid op v^+ E^+ \mid (if E^+ e e) \mid E^+ e \mid v^+ E^+ \mid (begin E^+ e) \\
\\
\frac{\Delta \Vdash E^+ : \Delta \quad \Delta \Vdash e_f \quad \Delta \Vdash e_s}{\Delta \vdash E^+ [(begin e_f e_s)] R_m (begin e_f (E^+ [e_s]))} \text{ M.6} \\
\frac{\Delta \Vdash E^+ : \Delta \quad \Delta \Vdash e_c \quad \Delta \Vdash e_f \quad \Delta \Vdash e_t}{\Delta \vdash E^+ [(if e_c e_f e_s)] R_m (if e_c (E^+ [e_f]) (E^+ [e_s]))} \text{ M.7} \\
\frac{\Delta, x \Vdash e_t \quad \Delta, x \Vdash e_f}{\Delta \vdash (if x e_t e_f) R_m (if x e_t (e_f [false/x]))} \text{ M.8} \\
\frac{\Delta \Vdash e_c \quad \Delta \Vdash e_t \quad \Delta \Vdash e_f}{\Delta \vdash (if (if e_c true false) e_t e_f) R_m (if e_c e_t e_f)} \text{ M.9} \\
\frac{\Delta \Vdash e_1 \quad \Delta \Vdash e_2 \quad \Delta \Vdash e_3}{\Delta \vdash (if (x = n_1) e_1 (if (x = n_2) e_1 e_3)) R_m (if (x = n_2) e_1 (if (x = n_1) e_1 e_3))} \text{ M.10}
\end{array}$$

Fig. 10. Additional  $R_m$  rules

- M.10 empowers the compiler to change the order of equality checks in nested conditional expressions when the branches involved are syntactically identical. Such rearrangements of checks can model the behavior of transformations of switch statements in the LLVM IR.

To establish the correctness of  $\rightarrow_m$ , we demonstrate that each of its rules preserves contextual equivalence (as induced by the standard reduction of the calculus). We do so following the standard recipe that we discuss at the beginning of Section 4: we define a standard binary step-indexed logical relation for the calculus, we prove it sound with respect to contextual approximation, and then we use the logical relation to prove correct each rule of  $\rightarrow_m$ . Specifically, for all  $i \geq 0$ , we define value and expression logical approximation at step  $i$  to be  $\mathcal{V}_i^{\rightarrow_s}$  and  $\mathcal{E}_i^{\rightarrow_s}$ .

*Definition 5.1 (Standard Logical Relation).*

$$\begin{aligned}
\mathcal{V}_i^{\rightarrow_s} &= \{(\lambda x.e_1, \lambda x.e_2) \mid \forall j < i. \forall v_1 v_2. (v_1, v_2) \in \mathcal{V}_j^{\rightarrow_s} \Rightarrow (e_1 [v_1/x], e_2 [v_2/x]) \in \mathcal{E}_j^{\rightarrow_s}\} \\
&\quad \cup \{(c, c)\} \\
\mathcal{E}_i^{\rightarrow_s} &= \left\{ (e_1, e_2) \mid \forall j < i. \forall a_1. e_1 \rightarrow_s^j a_1 \Rightarrow \right. \\
&\quad \left. \exists a_2. e_2 \rightarrow_s^* a_2 \wedge \right. \\
&\quad \left. \left( (a_1, a_2) \in \mathcal{V}_{i-j}^{\rightarrow_s} \vee (a_1 \equiv a_2 \equiv \text{unreachable}) \vee (\exists k. a_1 \equiv a_2 \equiv \text{error}_k) \right) \right\}
\end{aligned}$$

The relations  $\mathcal{V}^{\rightarrow_s}$  and  $\mathcal{E}^{\rightarrow_s}$  are straight-forward adaptations of standard binary step-indexed logical relations for proving contextual equivalences in functional languages. According to  $\mathcal{V}^{\rightarrow_s}$ , base values, errors, and unreachable are only related to themselves. Two functions are related only when, given related arguments, they produce related results.  $\mathcal{E}^{\rightarrow_s}$  relates closed expressions  $e, e'$  as long as  $e$  approximates  $e'$  after at most  $i$  steps under the standard reduction of the calculus. As usual, the step indices guarantee that the relations are well-founded.



$\mathcal{E}^{\rightarrow s}$  relates only closed expressions for a given number of steps. To prove contextual approximation for a pair of expressions, a stronger statement that generalizes over open expressions and arbitrary number of steps is necessary. Hence, we define logical approximation  $\Delta \vdash e_1 \leq e_2$ :

*Definition 5.2 (Logical Approximation).* For all  $\Delta, e_1, e_2$  such that  $\Delta \Vdash e_1$  and  $\Delta \Vdash e_2$ , we say that  $e_1$  logically approximates  $e_2$ ,  $\Delta \vdash e_1 \leq e_2$ , iff

$$\forall i \geq 0. \forall \gamma \in \mathcal{G}_i^{\rightarrow s}[\Delta]. (\gamma_1(e_1), \gamma_2(e_2)) \in \mathcal{E}_i^{\rightarrow s}.$$

where  $\mathcal{G}_i^{\rightarrow s}[\Delta]$  is as in Section 4 except that it draws pairs of values from  $\mathcal{V}^{\rightarrow s}$ .

Due to the compatibility lemmas of the standard logical relation, logical approximation is sound with respect to contextual approximation. The complete proof is on page 28 of Appendix G.

LEMMA 5.3 (SOUNDNESS). *If  $\Delta \vdash e \leq e'$  and  $\cdot \Vdash C : \Delta$  then*

- $UNDEF(C[e]) \implies UNDEF(C[e']),$
- $\forall c. C[e] \rightarrow_s^* c \implies C[e'] \rightarrow_s^* c,$
- $(\exists e_1. C[e] \rightarrow_s^* \lambda x. e_1) \implies (\exists e'_1. C[e'] \rightarrow_s^* \lambda x. e'_1),$  and
- $\forall k. C[e] \rightarrow_s^* \text{error}_k \implies C[e'] \rightarrow_s^* \text{error}_k.$

Finally, given the soundness of the standard logical relation, we prove that the  $\rightarrow_m$  reductions are correct by demonstrating that all expressions before and after each reduction logically approximate each other. For instance, to prove M.1 is correct, we show:

- If  $v \neq \text{false}$ ,  $\Delta \Vdash e_t$  and  $\Delta \Vdash e_f$  then  $\Delta \vdash e_t \leq (\text{if } v \text{ } e_t \text{ } e_f)$
- If  $v \neq \text{false}$ ,  $\Delta \Vdash e_t$  and  $\Delta \Vdash e_f$  then  $\Delta \vdash (\text{if } v \text{ } e_t \text{ } e_f) \leq e_t.$

The complete proof can be found on page 14 of Appendix H, where the proof for other  $R_m$  rules are also located.

## 6 BEYOND UNREACHABLE CODE

Beyond simple unreachability, unreachable can model a broad range of undefined behaviors, including division by zero, arithmetic overflow and under flow, and null pointer dereferencing. In particular, we can use unreachable to encode undefined behaviors due to uses of primitive operators with an illegal argument. For instance, integer division is well-defined over all integers unless the divisor is zero. Therefore, a compiler can assume that the erroneous divisor is never supplied to the operation. In other words, calls to division with a zero divisor are unreachable.

Abstractly, if some operation  $\mathcal{P}$  is undefined over some portion of its domain  $\mathcal{X}_{undef}$ , then we can encode such an operation as a conditional wrapper of the raw operation  $\mathcal{P}_r$ :

$$\mathcal{P} = \lambda x. (\text{if } (x \in \mathcal{X}_{undef}) \text{ unreachable } (\mathcal{P}_r x))$$

As a result, with an  $\rightarrow_u$  reduction, a call  $(\mathcal{P} x)$  reduces to  $(\mathcal{P}_r x)$  only if  $x \notin \mathcal{X}_{undef}$ . In essence, the insertion of unreachable signals to the compiler that the undefined behavior never occurs, and therefore, the compiler is allowed to optimize away the checks, exposing the raw operation. The beauty of this encoding, however, is that it other transformations that match the kinds of optimizations that compilers do, but without needing any new unreachable-specific rules.

To see how this captures more than simply eliminating the checks, we explore the undefined behavior of signed integer addition. According to the C standard ([International Organization for Standardization 2011](#)), “If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type the behavior is undefined”. Put differently, if the result of signed integer addition would be greater than INT\_MAX or smaller than INT\_MIN, then the addition exhibits undefined behavior. For instance, consider the C code snippet  $x < x + 1$ . If a C compiler decides to implement  $+$  by deferring to the underlying machine arithmetic,

and that arithmetic is two's complement, then the code snippet is equivalent to  $x == \text{INT\_MAX}$ . If, however, the compiler decides that overflow is undefined behavior, and therefore assuming it assumes that overflow never happen, then it is justified to compile to a constant expression that is always true. Indeed, gcc v7.5 (with the default options) produces code semantically (but not syntactically) equivalent to comparing  $x$  with  $\text{INT\_MAX}$ , but gcc v8.1 (with the default options) compiles the snippet to 1.

Our calculus can capture both of these possibilities. To do so, the calculus is extended with primitive operators,  $=_{\mathbb{Z}}$ ,  $\neq_{\mathbb{Z}}$ ,  $<_{\mathbb{Z}}$  and  $+_{\mathbb{Z}}$ , where each operator corresponds to its mathematical counterpart, which is well-defined for all (mathematical) integers. Additionally,  $x +_{\text{int}} y$ , which is akin to C's signed integer addition, is a shorthand for the conditional expression

$$\begin{aligned} &(\text{if } (\text{MAX}_{\text{int}} <_{\mathbb{Z}} x +_{\mathbb{Z}} y) \\ &\quad \text{unreachable} \\ &\quad (\text{if } (x +_{\mathbb{Z}} y <_{\mathbb{Z}} \text{MIN}_{\text{int}}) \\ &\quad \quad \text{unreachable} \\ &\quad \quad (x +_{\mathbb{Z}} y))) \end{aligned}$$

Given these operators, our calculus is able to equate the expression  $x <_{\mathbb{Z}} (x +_{\text{int}} 1)$  with both the expressions  $x \neq_{\mathbb{Z}} \text{MAX}_{\text{int}}$  and true. First, the unreachable calculus can equate  $x <_{\mathbb{Z}} (x +_{\text{int}} 1)$  with true by eliminating unreachables in  $x +_{\text{int}} 1$  with  $\rightarrow_u$  reductions. Second, the calculus can equate  $x <_{\mathbb{Z}} (x +_{\text{int}} 1)$  with  $x \neq_{\mathbb{Z}} \text{MAX}_{\text{int}}$  with successive applications of  $\rightarrow_m$  reductions. The basic idea is that an  $\rightarrow_m$  reduction can perform a reverse standard reduction to introduce a conditional that checks whether  $x$  is  $\text{MAX}_{\text{int}}$  or not. This transformation enables further  $\rightarrow_m$  reductions in each of the two branches taking advantage of the assumption that the result of the conditional's check is different for the then and else branch. The complete proof can be found on Appendix K.

## 7 UNREACHABLE IN RACKET ON CHEZ

The simplicity of the unreachable calculus raises the question of its relation to production compilers. In this section, we look at the relation between the reduction rules of our calculus and the way Racket's compiler transforms unreachable code.

Racket's core compiler is Chez Scheme. Overall, Chez Scheme iterates through a sequence of source-to-source passes a configurable number of times (the default is two). Chez Scheme's support for unreachable is primarily in the `cptypes` pass, which performs source-to-source optimizations based on type inference. In addition, transformations that involve simplifications for unused variables or expressions are also part of the earlier `cp0` source-to-source pass.

The notions of reduction in Figure 6 and Figure 5 map to specific lines in the implementation of Chez Scheme (the corresponding full files are part of the supplemental Appendix M):

Rules U.1 and U.2 correspond directly to the case that matches `if` forms in the implementations of the `cptypes` pass. Specifically, the `cptypes` pass optimizes recursively both branches of an `if` form. Together with the optimized branches, the pass returns additional information, including a type. If the pass determines that one of the branches never returns then the corresponding type is 'bottom'. The pass uses the `unsafe-unreachable?` predicate to identify a non-returning branch due to unreachable, and then replaces the `if` form with a call to `make-seq`, which constructs a `seq` expression, the same as our calculus's `begin` expression. Hence, the pass eliminates the branches of a conditional that do not terminate because they are unreachable in exactly the same way that the U.1 and U.2 rules simplify conditionals.

Rule P.1 corresponds to uses of the `make-seq` function. In detail, the `cp0` pass uses `make-seq` to optimize `seq` forms; it acts as a smart constructor that, same as rule P.1, drops from a sequence those expressions that are "simple" and whose result is unused. There's also a limited variant of the same

simplification in the `cptypes` pass; the duplication aims to reduce the number of optimizer-pass iterations needed in practice.

Rule P.2 corresponds to a case that matches `seq` forms in the `cptypes` pass. This case handles all expressions at the beginning of a `seq` form that do not return. Specifically, if the first expression of a `seq` does not return, then all subsequent expressions are ignored:

```
(define-pass cptypes ....
  [(seq ,e1 ,e2)
   (let-values ([ (e1 ty1 ....) (recur e1 ....)]
                [(e2 ty2 ....) (recur e2 ....)])
     (cond
      [(predicate-implies? ty1 'bottom) (unwrapped-error .... e1)]
      [else (values (make-seq .... e1 e2) ty ....)])])) ...)
```

In the code snippet, the call to `unwrapped-error` aims to adjust `e1` in some contexts to preserve non-tail positioning. The adjustment does not apply to `unreachable`. Hence, same as rule P.2, the pass replaces the `seq` form with `unreachable`.

Rule P.3 corresponds to how the `cp0` pass simplifies code with unused bindings using `begin` forms. Specifically, a loop of the `letify` function of the `cp0` pass gathers unused bindings and uses `residualize-seq` to lift their right-hand-side expressions to an enclosing sequence.

Rules P.4 and P.5 correspond to the `fold-call/other` function of the `cptypes` pass. The function recursively optimizes a list containing the function and argument expressions of an application. If, after the recursive optimizations, any of these sub-expressions has type `'bottom`, the application is replaced by that non-returning sub-expression. Unlike the calculus that has a fixed order of evaluation, order of evaluation in `Chez Scheme` is unspecified. As a result, the pass can treat all other sub-expressions as being downstream the non-returning sub-expression. Therefore, it discards them similar to the way rules P.4 and P.5 simplify applications in the calculus.

In sum, the rules of the calculus accurately describe the five places in the source of the Racket compiler that take advantage of `unreachable` for optimizations.

## 8 UNREACHABLE IN LLVM

The LLVM Intermediate Representation (LLVM IR) represents programs as control-flow graphs (CFGs). Unlike expression-based languages, CFGs break down programs into basic blocks that each consists of a linear sequence of instructions. Transfer of control between basic blocks is dictated by the edges of the CFG.

As such, compile-time transformations are no longer transformations of the structure of expressions but transformations of the structure of the CFG. Despite this difference, CFG transformations due to `unreachable` follow similar intuitions as those in our calculus. Since the `unreachable` is never executed, the LLVM compiler can prune any branches of control transfer operations that lead to `unreachable` and erase code preceding `unreachable`.

In this section, we establish a connection between these transformations and our calculus. Specifically, we prove a function that performs those two transformations correct, using the  $\rightarrow_c$  reductions of our calculus.

### 8.1 unreachable Transformations in LLVM, Informally

LLVM uses static single-assignment (SSA) form, thus a basic block in LLVM starts with a (possibly empty) sequence of  $\phi$  nodes that assign different values to variables depending on the predecessor executed at run time. The  $\phi$  nodes are followed by a series of instructions that typically define variables using the result of their computation. The last instruction of a basic block is called a terminator designating where the control should transfer to afterwards. Examples of terminators

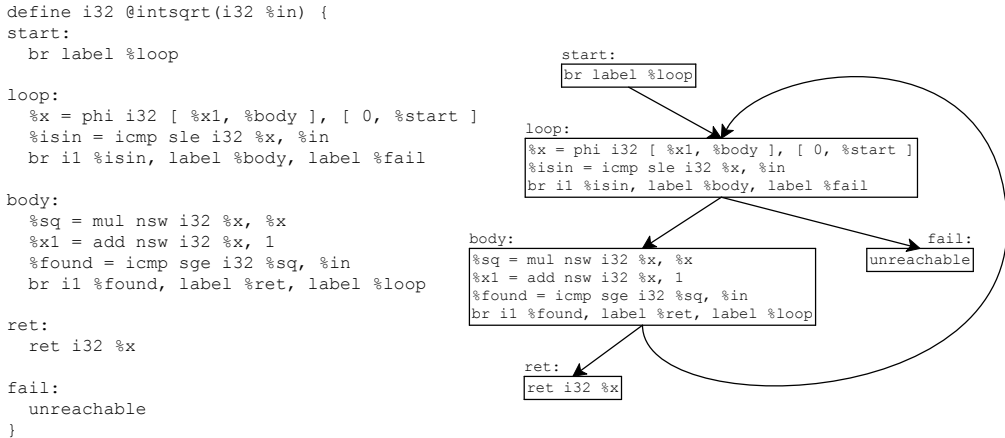


Fig. 11. LLVM Code Illustrating the Control Flow Graph Representation

in LLVM include a return statement, an unconditional branch, a conditional branch, and the unreachable instruction.

As a concrete example, figure 11 contains an LLVM IR function, `@intsqrt`, and its CFG. The `@intsqrt` function loops through the natural numbers, returning the first integer that is no smaller than the square root of `%in`. The block `loop:` starts with a  $\phi$  node that assigns a value to variable `%x`, the loop induction variable, counting up through the naturals. If the run-time predecessor of block `loop:` is block `start:`, then `%x` is 0; if the predecessor is block `body:` (following the long curved back edge), `%x` becomes equal to `%x1`, which will hold `%x + 1`. Subsequently, the loop checks the exit condition to determine whether `%x <= %in`, and the `br` terminator either transfers control to the body of loop or to `fail:`. Note that the loop increments `%x`, and the annotation `nsw` signals that no overflow happens,<sup>3</sup> hence, control transfers to `fail:` only if the input is negative.

As the block `fail:` contains the `unreachable` instruction, LLVM assumes that the block is never executed. Thus, LLVM removes it and its incoming edge, leaving only an *unconditional* `br` instruction in `loop:`. This simplification allows LLVM to erase the now-dead `%isin` definition, and merge `loop:` and `body:` since they are each other's unique predecessor and successor.

## 8.2 unreachable Transformations in Extended Vminus

We designed a transformation inspired by the optimization passes added along with the unreachable instruction to the LLVM (LLVM Project 2021) codebase in 2004.<sup>4</sup>

That code, upon spotting an unreachable terminator in the current block, erases all preceding instructions and prunes all the incoming edges, and so our transformation follows suit.

At the same time, other commits<sup>5</sup> to LLVM added the ability to replace instructions that obviously cannot be reached with `unreachable`. Our transformation does not capture this second capability of LLVM.

<sup>3</sup>More precisely, signed overflow would produce a poison value.

<sup>4</sup><https://github.com/llvm/llvm-project/commit/5edb2f32d00d39f7d9fd98b90ff440b5dbbdc45>

<sup>5</sup><https://github.com/llvm/llvm-project/commit/8ba9ec9bbb6625b149ae1ceeb46876553cd2f11> and <https://github.com/llvm/llvm-project/commit/a67dd32004bcc1a0a6fa2f0342e584187f5a403d>

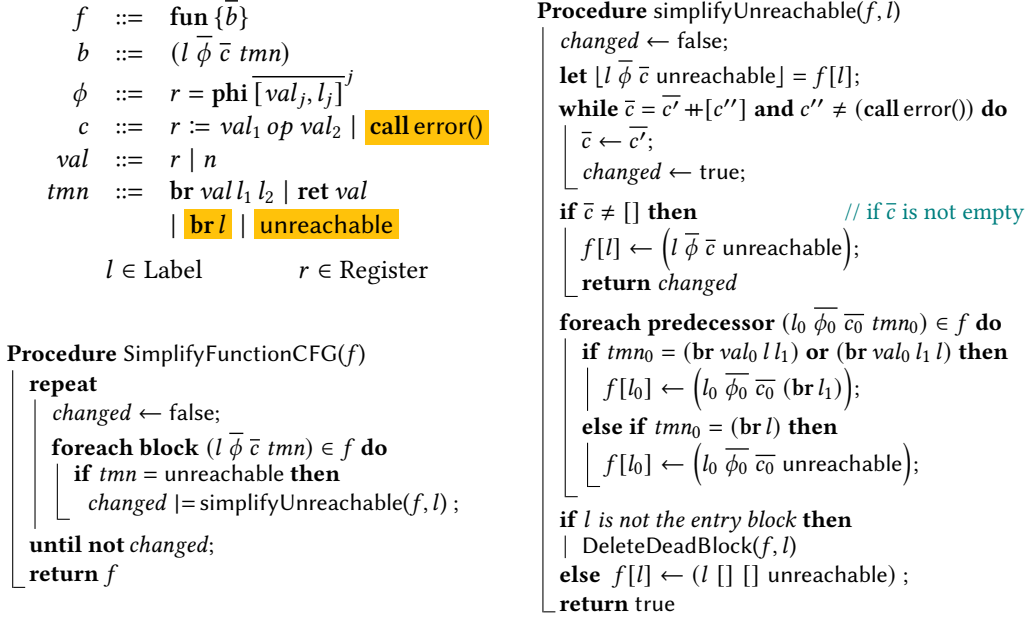


Fig. 12: The syntax of Extended Vminus and the unreachable transformations

Figure 12 shows our transformation. We extend the syntax of Vminus (Zhao et al. 2013) with additional terminators and commands highlighted in yellow.<sup>6</sup> Vminus is a minimal model of LLVM IR for studying SSA-based optimizations. While Vminus omits features like memory access and function calls, it is sufficient for our purposes since LLVM primarily transforms unreachable instructions by rearranging basic blocks and erasing commands.

A program in Vminus is a function ( $f$ ) that contains a list of basic blocks ( $\bar{b}$ ). Each basic block is a 4-tuple consisting of a label ( $l$ ), a list of  $\phi$  nodes ( $\bar{\phi}$ ), a list of commands ( $\bar{c}$ ) and a terminator ( $tmn$ ). A command  $c$  either assigns the result of a binary operation to a fresh variable or calls the error function to end the execution. We use `call error()` to model commands that *do not* transfer control to the next command in the basic block.<sup>7</sup> Following Zhao et al. (2013)'s notation, an overlined non-terminal represents a list of the underlying non-terminal. The notation  $l.i$  denotes the  $i$ -th command in block  $l$ , and  $f[l] = [b]$  represents a look up of the block labeled  $l$  in  $f$ . The notation  $[b]$  asserts that the lookup succeeds with the result  $b$ .

`SimplifyFunctionCFG( $f$ )` is the entry point of our transformation. It iteratively updates  $f$  until reaching a fixed point. In each iteration, `SimplifyFunctionCFG` scans through the basic blocks to identify basic blocks terminating with `unreachable`. For any such basic block  $l$ , `SimplifyFunctionCFG` invokes `simplifyUnreachable` to remove commands and paths leading up to  $l$ 's `unreachable`. This includes simplifying the commands in block  $l$  and possibly the branch instructions in the predecessors of  $l$ .

The `simplifyUnreachable( $f, l$ )` transformation exploits the intuition that an unreachable instruction should never be reached; otherwise the behavior of the program is undefined. Specifically, `simplifyUnreachable` does two simplifications. First, it scans the commands in  $l$  from the end of the

<sup>6</sup>We also omit the type annotations and use integers as the only constant values.

<sup>7</sup>`call error()` terminates the program much like the `exit` function. It is unrelated to exceptions.

$$\begin{aligned}
\mathcal{H}_{proc}(f) &= \lambda x. (\mathcal{H}_{jump}(f, l_0) \ 0) \quad \text{where } x \text{ is fresh and } l_0 \text{ is the label of the entry block} \\
\mathcal{H}_{jump}(f, l) &= \lambda r_1 \dots r_n. \mathcal{H}_{cs}(f, l, \bar{c}) \\
&\quad \text{where } f[l] = [l \ \bar{\phi} \ \bar{c} \ tmn] \text{ and } \bar{\phi} = (r_1 = \mathbf{phi} [\overline{val_{j_1}, l_{j_1}}]^{j_1}) \dots (r_n = \mathbf{phi} [\overline{val_{j_n}, l_{j_n}}]^{j_n}) \\
&\quad \text{if } \bar{\phi} = [], \text{ we introduce a dummy fresh variable } r_0 \\
\mathcal{H}_{cs} : f \ l \ \bar{c} &\longrightarrow e \\
\mathcal{H}_{cs}(f, l, ((\mathbf{call\ error}()), \bar{c}')) &= (\mathbf{begin\ error} \ \mathcal{H}_{cs}(f, l, \bar{c}')) \\
\mathcal{H}_{cs}(f, l, ((r := val_1 \ op \ val_2), \bar{c}')) &= (\mathbf{let} \ ([r \ (op \ val_1 \ val_2)]) \ \mathcal{H}_{cs}(f, l, \bar{c}')) \\
\mathcal{H}_{cs}(f, l, []) &= (\mathbf{letrec} \ ([l_1 \ \mathcal{H}_{jump}(f, l_1)] \dots [l_m \ \mathcal{H}_{jump}(f, l_m)]) \\
&\quad \mathcal{H}_{term}(f, l)) \\
&\quad \text{where } l_1 \dots l_m \text{ are the children of node } l \text{ in the dominator tree} \\
\mathcal{H}_{term} : f \ l &\longrightarrow e \\
\mathcal{H}_{term}(f, l) &= val \quad \text{if } f[l] = [l \ \bar{\phi} \ \bar{c} \ (\mathbf{ret} \ val)] \\
\mathcal{H}_{term}(f, l) &= \mathbf{unreachable} \quad \text{if } f[l] = [l \ \bar{\phi} \ \bar{c} \ \mathbf{unreachable}] \\
\mathcal{H}_{term}(f, l) &= (l' \ \overline{val'}) \quad \text{if } f[l] = [l \ \bar{\phi} \ \bar{c} \ (\mathbf{br} \ l')] \\
&\quad \text{where } f[l'] = [l' \ \bar{\phi}' \ \bar{c}' \ tmn'], \ \overline{getIncomingValueForBlock(\phi', l)} = \overline{val'} \\
&\quad \text{when } \bar{\phi}' = [], \text{ we supply a dummy argument } 0 \\
\mathcal{H}_{term}(f, l) &= (\mathbf{if} \ val \ (l' \ \overline{val'}) \ (l'' \ \overline{val''})) \quad \text{if } f[l] = [l \ \bar{\phi} \ \bar{c} \ (\mathbf{br} \ val' \ l'')] \\
&\quad \text{where } f[l'] = [l' \ \bar{\phi}' \ \bar{c}' \ tmn'], \ \overline{getIncomingValueForBlock(\phi', l)} = \overline{val'} \\
&\quad \quad f[l''] = [l'' \ \bar{\phi}'' \ \bar{c}'' \ tmn''], \ \overline{getIncomingValueForBlock(\phi'', l)} = \overline{val''} \\
&\quad \text{we supply } 0 \text{ as a dummy argument if there are no } \phi \text{ nodes as in the case for } (\mathbf{br} \ l') \\
\overline{getIncomingValueForBlock} : \phi \ l &\longrightarrow val \\
\overline{getIncomingValueForBlock}(\phi, l) &= val_i \text{ where } \phi = (r = \mathbf{phi} [val_1, l_1] \dots [val_n, l_n]) \text{ and } l = l_i
\end{aligned}$$

Fig. 13: Translating Extended Vminus Functions to the Unreachable Calculus

list. If the scanned command ( $c''$ ) *always* transfers control to the next instruction in  $l$ , it is erased from  $l$ . The scan ends when `simplifyUnreachable` hits a command that *may not* transfer control the next command. Second, `simplifyUnreachable` prunes the incoming edges. For each predecessor  $l_0$  of  $l$ , `simplifyUnreachable` removes  $l$  from the branch targets of  $l_0$ . If  $l_0$ 's terminator is an unconditional branch, `simplifyUnreachable` replaces it with `unreachable`.

### 8.3 Translating Extended Vminus to The Unreachable Calculus

The transformation from section 8.2 is based on the same insights about unreachable as the  $\rightarrow_u$  and  $\rightarrow_p$  reductions of our calculus. As such, it should be correct despite the fact that it belongs to a different linguistic setting than that of the calculus.

To use the calculus to prove that it is correct, we leverage [Kelsey \(1995\)](#)'s algorithm that translates programs from SSA to  $\lambda$  expressions and back. In detail, we adapt their algorithm to work on Extended Vminus programs and use it to show that the transformation in fig. 12 given a program A produces program B such that the translation of A is equal to the translation of B under  $\rightarrow_c$  reductions.

fig. 13 gives the complete definition of the translation.  $\mathcal{H}_{proc}$  translates a function  $f$  in Extended Vminus to a (closed) function  $\lambda x. e$  in the unreachable calculus. It comprises three auxiliary functions:  $\mathcal{H}_{jump}$  translates a basic block to a  $\lambda$  function,  $\mathcal{H}_{cs}$  takes a list of commands and morally translates

them into a nested let expression, and finally  $\mathcal{H}_{term}$  translates the terminator of a block into an appropriate expression in our calculus. In the definition of  $\mathcal{H}_{cs}$ ,  $(\text{let } ([x\ e_1])\ e_2)$  is a syntactic sugar for  $(\lambda x.e_2)\ e_1$  and `letrec` is implemented using the Y combinator. Appendix L details the implementation.

The key idea behind  $\mathcal{H}_{proc}$  is the encoding of the CFG of a given program.  $\mathcal{H}_{jump}$  encodes each basic block as a single function, while  $\mathcal{H}_{term}$  adds edges that represent branches using function applications. For each basic block handled by  $\mathcal{H}_{jump}$ , the variables defined by the  $\phi$  nodes become the formal parameters of the resulting function, and the incoming values of the  $\phi$  nodes are turned into the arguments in the function applications that  $\mathcal{H}_{term}$  uses to encode branch instructions.

Having translated individual basic block and the edges of the CFG, the translation assembles the results of  $\mathcal{H}_{jump}$  into a single expression while respecting the variable scoping rules of Vminus programs. To ensure that variables are defined before they are referenced, Vminus requires a variable definition to appear in a block that dominates all the blocks that use the variable. Intuitively, block  $l$  *dominates* block  $l'$ , written as  $l \succcurlyeq l'$ , if block  $l$  appears in every path from the entry to block  $l'$ . The dominance relation between basic blocks form a so-called *dominator tree* (Lengauer and Tarjan 1979; Lowry and Medlock 1969): if block  $l$  dominates  $l'$  then  $l$  is an ancestor of  $l'$  in the dominator tree. Conversely, if  $l$  has children  $l_1, \dots, l_m$  in the dominator tree, then  $l$  dominates all  $l_i$  and each function  $\mathcal{H}_{jump}(f, l_i)$  should be able to reference the variables that block  $l$  defines. Therefore,  $\mathcal{H}_{cs}$  arranges the results of  $\mathcal{H}_{jump}$  into nested `letrec`s in accordance with the dominator tree of the control flow graph to preserve correct scoping of the variables.

With the definition of the translation in hand, we can prove the correctness of its correctness by establishing the correctness of `simplifyUnreachable`:

**THEOREM 8.1.** *Let  $f[l] = [l\ \bar{\phi}\ \bar{c}\ \text{unreachable}]$  and  $f'$  be the new function after running the transformation `simplifyUnreachable`( $f, l$ ). If  $l$  is reachable from the entry point of  $f$  then  $\mathcal{H}_{proc}(f) \rightarrow_c^* \mathcal{H}_{proc}(f')$ .*

**PROOF.** Because block  $l$  is reachable from the entry point of  $f$  and has no successor, it must be a leaf node in the dominator tree. Thus,  $\mathcal{H}_{cs}(f, l, \bar{c})$  is a subexpression of  $\mathcal{H}_{proc}(f)$ . To analyze how `simplifyUnreachable` changes  $f$ , we take cases on whether  $\bar{c}$  contains `call error()` or not.

If  $\bar{c}$  does not contain `call error()`, `simplifyUnreachable` prunes all incoming edges of block  $l$  and deletes the entire block from  $f$ . Let  $f[l_0] = [l_0\ \bar{\phi}_0\ \bar{c}_0\ \text{tmn}_0]$  be any predecessor of  $l$ . If  $\text{tmn}_0 = (\mathbf{br}\ \text{val}_0\ l\ l_1)$ , its translation is  $\mathcal{H}_{term}(f, l_0) = (\text{if}\ \text{val}_0\ (l\ \text{val})\ (l_1\ \text{val}_1))$  where the arguments are extracted from the  $\phi$  nodes in block  $l$  and  $l_1$ .

By inlining the translation of block  $l$  with a series of  $\rightarrow_c$  reductions, we obtain the expression  $(\text{if}\ \text{val}_0\ (\mathcal{H}_{jump}(f, l)\ \bar{\text{val}})\ (l_1\ \bar{\text{val}}_1))$ . However, the body of  $\mathcal{H}_{jump}(f, l)$ ,  $\mathcal{H}_{cs}(f, l, \bar{c})$ , reduces to `unreachable` under  $\rightarrow_c$  since it is a nested `let` expression whose body,  $\mathcal{H}_{term}(f, l)$ , is `unreachable`. Thus the entire `if` expression simplifies to  $(l_1\ \bar{\text{val}}_1)$ , which is precisely the translation of  $(\mathbf{br}\ l_1)$ .

The case where  $\text{tmn}_0 = (\mathbf{br}\ l)$  is also similar.

Finally, after all predecessors of  $l$  are updated,  $\mathcal{H}_{jump}(f, l)$  has no reference and thus its binding can be dropped, resulting  $\mathcal{H}_{proc}(f')$ .

In this proof, we have used several `letrec` identities such as inlining a definition and dropping an unreferenced binding. Appendix L proves these identities using the  $\rightarrow_m$  rules from Figure 9.

The case where  $\bar{c}$  includes `call error()` is similar to the simplification of  $\mathcal{H}_{cs}(f, l, \bar{c})$  in the previous case except that `unreachable` stops erasing the `let` bindings after reaching the expression `error`. Say  $\bar{c}$  equals  $\bar{c}' \# [\text{call error}()] \# \bar{c}''$  such that  $\bar{c}'$  contains no `call error()` commands, we know that `simplifyUnreachable`( $f, l$ ) changes block  $l$  to  $(l\ \bar{\phi}\ (\bar{c}' \# [\text{call error}()])\ \text{unreachable})$ . Therefore we



need to prove

$$\mathcal{H}_{cs}(f, l, \overline{c'} \# [\text{call error}()] \# \overline{c''}) \rightarrow_c^* \mathcal{H}_{cs}(f', l, \overline{c'} \# [\text{call error}()]).$$

Note that  $\mathcal{H}_{term}(f, l) = \mathcal{H}_{term}(f', l) = \text{unreachable}$ , so this is straightforward as  $\mathcal{H}_{cs}$  translates  $\overline{c''}$  to a nested let expression whose body is just unreachable. Thus  $\text{simplifyUnreachable}$  preserves the behavior of the program.  $\square$

As a final remark in this section, the approach to proving CFG transformations correct via translation to the  $\lambda$  calculus does not scale to proving realistic compilers correct. Of course, this is not the goal of this section but we discuss it here to eliminate any confusion. Beyond the obvious shortcoming that the  $\lambda$  calculus and imperative features are not well-aligned, there is an additional and subtle technical challenge at play. While the reductions of the unreachable calculus leave the overall structure of expressions unchanged, unreachable transformations in Extended Vminus, and LLVM, can modify the dominator tree of a program in complex ways. Hence, relating the result of a series of reductions with the result of an unreachable transformations in Extended Vminus requires equating expressions with arbitrarily different structure. Put differently, the compile-time semantics of the unreachable calculus do not map directly to the unreachable transformations in Extended Vminus, at least via translations, like Kelsey's, that depend on the dominator tree of the input program. We conjecture that this discrepancy is due to the translation, but we cannot exclude a misalignment between the unreachable calculus and the CFG-based world of Extended Vminus and LLVM.

## 9 RELATED WORK

Our work is the first that develops an equational theory for unreachable.

Other techniques that examine the correctness of compilers have to also deal, one way or another, with the semantics of unreachable and other undefined behaviors. CompCert (Leroy 2009a,b) gives semantics to undefined behavior implicitly, by specifying defined behavior with a co-inductive structure. Based on this structure, the CompCert project proves correct whole-program transformations in a realistic C compiler. While the linguistic setting and the scale of CompCert are not comparable with this work, the reductions of our calculus equate (open) expressions in all contexts rather than whole programs. Furthermore, the correctness of our reductions assumes the intuitive predicate  $\neg \text{UNDEF}$ , instead of CompCert's co-inductive definition of defined behavior.

Similar to CompCert, Jung et al. (2020) give a definition for defined behavior in Rust, namely the Stacked Borrows pattern. Any program that violates this pattern is considered to exhibit undefined behavior. The authors show that Stacked Borrows is sufficient to validate optimizations in the Rust compiler involving both safe and unsafe code, and that it admits realistic Rust programs. Instead of an approximate predicate, our calculus relies on a precise definition of undefined behavior.

Vellvm (Zakowski et al. 2021; Zhao et al. 2012, 2013) is a long-running project for the formal verification of transformations in LLVM. It covers various versions of undefined behavior, including unreachable, by reducing them to a basic notion of undefined behavior similar to the discussion in section 6. The latest version of Vellvm relies on interaction trees (Xia et al. 2020), and we conjecture it can prove correct equivalences that correspond to the compile-time reductions of our calculus. However, such proofs would need to establish equalities between denotations of LLVM code fragments, rather than the syntax-based equivalences of our calculus, which we claim match the way compiler writers reason about code through local rewriting steps.

Dahiya and Bansal (2017) presents a simulation relation for C programs that takes undefined behavior into account. Their work considers a number of different forms of undefined behavior, but not unreachable.



Mears (2021)'s proposals favor the addition of a construct like unreachable to C and C++. The authors note the advantages of introducing such a construct for optimizations. Similar to Racket and Rust, the authors point out that for debugging purposes, the proposed construct could be treated as an exception at run time.

As a final note, a considerable body of work focuses on program checkers that detect undefined behavior, including unreachable. Some checkers rely on static analysis (Dietz et al. 2012; Jourdan et al. 2015; Wang et al. 2016), while others on testing (Regehr 2011). Here we focus on two of these works. Hathhorn et al. (2015) describes a model checker that detects undefined behavior based on a formal semantics for undefined behavior in C. RustBelt (Jung et al. 2018) proves the absence of one kind of undefined behavior from Rust programs, data races. It relies on semantic type soundness, which admits programs that the conventional syntactic type soundness rejects. While all these techniques define what certain kinds of undefined behavior mean in different settings, they seek to eliminate unexpected undefined behavior as opposed to explaining the optimization opportunities that undefined behavior provides.

## 10 CONCLUSION

This paper gives a formal account of the essence of unreachable. Specifically, it confronts head on that unreachable is a form of undefined behavior, and hence, a compiler may take advantage of it to legally transform a program in a way that causes it to evaluate differently. For that reason, the paper presents a pair of specifications for a programming language: one that covers how programs evaluate normally and a separately-defined and strikingly simple one that covers what transformations are legal for a compiler. We prove that, despite its simplicity, the specification of the compiler is correct: its rules preserve the meaning of programs (according to normal evaluation), under the assumption that the original programs do not exhibit undefined behavior, i.e., they do not evaluate unreachable. Importantly, the correctness of the compile-time transformations depends on this precise definition of undefined behavior, rather than an approximation that aims to facilitate the proofs. In other words, our formal specification of unreachable provides simple rewriting rules that capture how compiler writers reason about undefined behavior and how they use this reasoning to justify transformation implementations.

Taking a step back, we hope that our approach can provide a template for others who wish to formally state and prove meta-theoretic properties that correctly capture aspects of undefined behavior. While undefined behavior has a reputation as an unruly phenomenon, our work shows that there are well-behaved undefined behaviors. Similar to unreachable, we conjecture that other aspects of undefined behavior can be described precisely and intuitively. Hence, we see this work as the first step towards demystifying undefined behavior.

## REFERENCES

- CVE-2014-0160. 2014. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160> The Heartbleed Bug. Retrieved: July, 2022. Discovered by Neel Mehta from Google.
- Manjeet Dahiya and Sorav Bansal. Modeling Undefined Behaviour Semantics for Checking Equivalence Across Compiler Optimizations. In *Proc. Haifa Verification Conference*, 2017. [https://doi.org/10.1007/978-3-319-70389-3\\_2](https://doi.org/10.1007/978-3-319-70389-3_2)
- Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding Integer Overflow in C/C++. In *Proc. International Conference on Software Engineering*, 2012. <https://doi.org/10.1109/ICSE.2012.6227142>
- Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. *ACM Conference on Programming Language Design and Implementation*, 2015. <https://doi.org/10.1145/2737924.2737979>
- International Organization for Standardization. ISO/IEC 14882:2011 C++ Standard. 2011. <https://www.iso.org/standard/50372.html>

- Jacques Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 247–259, 2015. <https://doi.org/10.1145/2676726.2676966>
- Ralf Jung. Undefined Behavior deserves a better reputation. 2021. <https://blog.sigplan.org/2021/11/18/undefined-behavior-deserves-a-better-reputation/> Retrieved: July, 2022.
- Ralf Jung, Hoanghai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages (POPL)* 4, pp. 41:1–41:32, 2020. <https://doi.org/10.1145/3371109>
- Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages (POPL)* 2, pp. 66:1–66:34, 2018. <https://doi.org/10.1145/3158154>
- Richard A. Kelsey. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Proc. Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pp. 13–22, 1995. <https://doi.org/10.1145/202530.202532>
- Thomas Lengauer and Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems* 1(1), pp. 121–141, 1979. <https://doi.org/10.1145/357062.357071>
- Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM* 52, pp. 7:107–7:115, 2009a. <https://doi.org/10.1145/1538788.1538814>
- Xavier Leroy. Mechanized Semantics for the C Subset of the C Language. *Journal of Automated Reasoning* 43, pp. 263–368, 2009b. <https://doi.org/10.1007/s10817-009-9148-3>
- LLVM Project. LLVM 13.0.0 Release Notes. 2021. <https://releases.llvm.org/13.0.0/docs/ReleaseNotes.html> Retrieved: Sep, 2021
- Edward S. Lowry and C. W. Medlock. Object Code Optimization. *Communications of the ACM* 12(1), pp. 13–22, 1969. <https://doi.org/10.1145/362835.362838>
- Melissa Mears. Function to mark unreachable code. 2021. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0627r6.pdf>
- Godon D. Plotkin. {Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science*(1(2)), pp. 125–159, 1975.
- John Regehr. Better Testing With Undefined Behavior Coverage. 2011. <https://blog.regehr.org/archives/388>
- Xi Wang, Nikolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. A Differential Approach to Undefined Behavior Detection. *Communications of the ACM* 59(3), pp. 99–106, 2016. <https://doi.org/10.1145/2885256>
- Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction Trees: rRepresenting Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages (POPL)* 4, pp. 51:1–51:32, 2020. <https://doi.org/10.1145/3371119>
- Yannick Zakowski, Calvin Beck, Irene Yoon, Iliia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proceedings of the ACM on Programming Languages (ICFP)* 5, pp. 67:1–67:30, 2021. <https://doi.org/10.1145/3473572>
- Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. ACM Symposium on Principles of Programming Languages*, POPL '12, pp. 427–440, 2012. <https://doi.org/10.1145/2103656.2103709>
- Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal Verification of SSA-Based Optimizations for LLVM. In *Proc. ACM Conference on Programming Language Design and Implementation*, PLDI '13, pp. 175–186, 2013. <https://doi.org/10.1145/2491956.2462164>