Safety in Programming Languages

Matthias Felleisen and Shriram Krishnamurthi

Rice COMP TR99-352
November 1999

Safety in Programming Languages

Matthias Felleisen and Shriram Krishnamurthi
Department of Computer Science
Rice University
Houston, Texas

Summary

People widely use the term "safe programming language," assuming it has an agreed-upon meaning, which is unfortunately not the case. Some people associate safety with the type system of a language; others relate it to the operational behavior of programs.

We formulate what we believe to be the folklore meaning of "safety" and explore the implications of this definition. Specifically, we model both a safe and an unsafe semantics for the same syntax. Using these models, we formulate theorems that succinctly characterize and compare typical safe and unsafe evaluators. The theorems also pinpoint the key advantages of safety.

In the second part of the paper, we investigate the meaning of Milner's type soundness theorem in the context of safe and unsafe languages. We show how the type soundness theorem holds for safe languages and that type soundness is meaningless for unsafe languages. In addition we extend the models with dynamic memory allocation and discuss how memory reclamation affects safety. More specifically, we sketch the semantics of a language with dynamic memory allocation, formulate a semantics of garbage collection and manual memory management, and explore the soundness and safety of both. Roughly speaking, a garbage collector's relationship to safety is analogous to that of a type system, but instead of guaranteeing that fewer run-time errors occur, it only decreases the likelihood of certain errors. As expected, manual memory management is unsound.

1 Programming and Safety

Mechanical engineers design fail-safe mechanisms and processes. These systems come to rest in the least-damaging position when any part of the device malfunctions. For example, the bars at a railroad crossing fall to the blocking position if both the primary power-supply and the backup battery fail to provide electricity.

If software engineers wish to produce minimally robust systems, they must first agree on what it means for programming languages, their primary tools, to be fail-safe. While computer scientists widely use the term "safe language," they don't agree on its precise meaning. Some associate safety with the type system of a language; others relate it to the operational behavior of programs.

Following the analogy with mechanical engineering, a program in a safe language should stop if one of its components malfunctions. The most basic components of a program are the atomic computational operations of the underlying language. An operation's purpose is to process data, *e.g.*, to add numbers or to use a boolean value to decide what to execute. Since the universe of data is larger than what an operation is typically intended to process, it can happen that an operation is mis-applied to data outside of the proper range.

Starting from this observation, we say that a language is safe (1) if it precisely specifies the set of data for which its operations are defined and (2) if it signals an error when an operation is applied to inappropriate data. The enforcement may consist of a combination of compile-time and run-time checks. We believe that this characterization is as close as possible to the folklore meaning of "safety" and explore its implications in this paper.

In the following section, we model a safe and an unsafe version for a small programming language. The theorems in this section characterize safety through a comparison of the two models and highlight the advantages of safe programming languages. The third section re-examines Milner's [10] type soundness theorem in the context of our investigation. The fourth section extends the investigation to a language with dynamic memory allocation and analyzes the roles of automatic and manual garbage collection. The final sections relate our work to that of others and discuss future uses of our work.

$$
\begin{array}{lrcl}
 & P & = & D;\ S;\ \mathsf{return}\ I \\
 & D & = & d; \ldots;\ d \\
 & d & = & I\ =\ V \mid I[V]\ =\ V \ldots V \\
 & S & = & \mathsf{skip} \mid I\ :=\ E \mid I[E]\ :=\ E \mid \{S; \ldots;\ S\} \mid \mathsf{if}\ (E)\ S\ \mathsf{else}\ S \mid \mathsf{while}\ (E)\ \mathsf{do}\ S \\
 & E & = & I \mid I[E] \mid\ \mid V \mid E{+}E \mid E{-}E \mid E{*}E \mid E/E \mid E{\leq}E \mid E{\geq}E \\
x, y, & I & = & \{x, y, z, \ldots\} \\
v, u & V & = & \mathbf{B} \mid \mathbf{Z} \\
b & \mathbf{B} & = & \mathsf{true} \mid \mathsf{false} \\
k, l & \mathbf{Z} & = & 0 \mid +1 \mid -1 \mid \ldots
\end{array}
$$

Figure 1: The Syntax of Language **A**

---

2 Safe Programming Languages

In this section we first lay out the basic definitions, including examples, and then define and explore safety in this context.

Syntax and Semantics: The definition of a programming language specifies its syntax and semantics. The former defines the alphabet, the vocabulary, and the grammatical rules, plus some

context-sensitive constraints. The latter explains what the phrases and elements of the syntax mean, or, how they are to be evaluated on a machine. The semantics makes a programming language safe or unsafe and specifies what to expect from safety.

To illustrate these concepts, we introduce **A**, a primitive programming language as a running example. Its syntax is specified in figure 1. An **A**-program consists of three parts: a sequence of definitions, a statement, and a return identifier. The latter determines the only visible result of a program. The definitions introduce individual identifiers and array identifiers with their initial values (numeric and boolean constants). By convention, the identifier *arg* is defined and stands for the program's only input: a number. A statement is either an assignment, an if-statement, a while-loop, or a compound statement. The first three also contain expressions, which are either constants, identifiers, array references, or some standard arithmetical and relational expressions.

Statements		
I := $\bullet$	V	
$I[\bullet_1]$:= $\bullet_2$	$[1,k]$	V
if ($\bullet$) S else T	**B**	
$\{\bullet; S_2; \ldots; S_n\}$	skip	

Expressions		
$I[\bullet_1]$	$[1,k]$	
$\bullet_1 + \bullet_2$	**Z**	**Z**
$\bullet_1 - \bullet_2$	**Z**	**Z**
$\bullet_1 * \bullet_2$	**Z**	**Z**
$\bullet_1 / \bullet_2$	**Z**	**Z**
$\bullet_1 \leq \bullet_2$	**Z**	**Z**
$\bullet_1 \geq \bullet_2$	**Z**	**Z**

Figure 2: Value-Strict Positions in Statements and Expressions of **A**

We assume that an **A**-program satisfies a small number of context-sensitive constraints:

1. All identifier occurrences in the program are bound by one of the global definitions, which in turn introduce mutually distinct identifiers.
2. An array definition must have the form $a[k]$:= $V_1 \ldots V_k$; that is, its length specification must be a number, say k, and the right-hand side must be a sequence of k constants.
3. The head identifier in an array assignment, $a[E]$:= E, is declared as an array.
4. The head identifier in an array dereference, $a[E]$, is also declared as an array.

Defining functions on the syntax of a program is a convenient way to specify the semantics of a language as an abstract machine. In this setting a program is a state, a function is an instruction, and mapping a program to its image corresponds to the execution of an instruction. The goal of executing the "instructions" is to determine the value that the program, or one of its components, produces. Since our machine is specified syntactically, we must first classify some phrases as *values*. We call the remaining ones (program) *operations*; the sub-phrases of an operation are often called *arguments*.

A language like **A** with multiple syntactic categories may contain values for each category. Indeed, **A** contains only one form of statement value: skip, but many different expression values: numbers and booleans. Conversely, with the exception of constants, all expressions are operations: identifiers, array reference, arithmetical and comparative expressions. Similarly, all statements, except for skip, are operations.

Before the machine can excute an operation, some or all of its sub-phrases must be reduced to values. We say that the operation is *strict* at the positions of these sub-phrases. When the values are available, the operation performs some action, which possibly involves the evaluation of other

Contexts:

$$
\begin{array}{lcl}
\mathcal{D}\langle\bullet\rangle & = & d; \ldots; \bullet; \ldots; d \\
\mathcal{S}\langle\bullet\rangle & = & \bullet \mid \{\mathcal{S}\langle\bullet\rangle; \ldots; S\} \\
\mathcal{E}\langle\bullet\rangle & = & \mathcal{S}\langle x := \mathcal{F}\langle\bullet\rangle\rangle \mid \mathcal{S}\langle x[\mathcal{F}\langle\bullet\rangle] := E\rangle \mid \mathcal{S}\langle x[V] := \mathcal{F}\langle\bullet\rangle\rangle \mid \mathcal{S}\langle\mathsf{if}\ (\mathcal{F}\langle\bullet\rangle)\ E\ \mathsf{else}\ E'\rangle \\
\mathcal{F}\langle\bullet\rangle & = & \bullet \mid I[\mathcal{F}\langle\bullet\rangle] \mid \mathcal{F}\langle\bullet\rangle{+}E \mid V{+}\mathcal{F}\langle\bullet\rangle \mid \ldots
\end{array}
$$

Core Transitions:

return:				
$\mathcal{D}\langle x = v\rangle$;	skip; return x	$\mapsto_p$	v	
identifier assignment:				
$\mathcal{D}\langle x = u\rangle$;	$\mathcal{S}\langle x := v\rangle$;	$\mapsto_p$	$\mathcal{D}\langle x = v\rangle$;	$\mathcal{S}\langle$skip$\rangle$;
identifier dereference:				
$\mathcal{D}\langle x = v\rangle$;	$\mathcal{E}\langle x\rangle$;	$\mapsto_p$	$\mathcal{D}\langle x = v\rangle$;	$\mathcal{E}\langle v\rangle$;
array assignment:				
$\mathcal{D}\langle x[k] = \ldots\ u_l\ \ldots\rangle$; if $1 \le l \le k$	$\mathcal{S}\langle x[l] := v\rangle$;	$\mapsto_p$	$\mathcal{D}\langle x[k] = \ldots\ v\ \ldots\rangle$;	$\mathcal{S}\langle$skip$\rangle$;
array dereference:				
$\mathcal{D}\langle x[k] = \ldots\ u_l\ \ldots\rangle$; if $1 \le l \le k$	$\mathcal{E}\langle x[l]\rangle$;	$\mapsto_p$	$\mathcal{D}\langle x[k] = \ldots\ u_l\ \ldots\rangle$;	$\mathcal{E}\langle u_l\rangle$;
begin:				
D;	$\mathcal{S}\langle\{$skip$; s_2; \ldots; s_n\}\rangle$;	$\mapsto_p$	D;	$\mathcal{S}\langle\{s_2; \ldots; s_n\}\rangle$;
skip:				
D;	$\mathcal{S}\langle\{$skip$\}\rangle$;	$\mapsto_p$	D;	$\mathcal{S}\langle$skip$\rangle$;
if-true:				
D;	$\mathcal{S}\langle$if (true) S else $T\rangle$;	$\mapsto_p$	D;	$\mathcal{S}\langle S\rangle$;
if-false:				
D;	$\mathcal{S}\langle$if (false) S else $T\rangle$;	$\mapsto_p$	D;	$\mathcal{S}\langle T\rangle$;
loop unrolling:				
D;	$\mathcal{S}\langle$while (E) do $S\rangle$;	$\mapsto_p$	D; if (E) $\{S;$ while (E) do $S\}$ else skip;	
expression reduction:				
D;	$\mathcal{E}\langle e\rangle$;	$\mapsto_p$	D; if $e \mapsto_e e'$	$\mathcal{E}\langle e'\rangle$;

Reductions of Expressions:

$1{+}1 \mapsto_e 2$ ... $2{-}1 \mapsto_e 1$ ... $1{*}1 \mapsto_e 1$ ... $1/2 \mapsto_e 1$ ...

No reductions for $n/0$

Figure 3: The Core Semantics of Language **A**

sub-phrases or the creation of new values. In general, some language operations are *partial*, i.e., they can only act on a proper subset of those values to which it can be applied. We refer to these subsets as the run-time types of the operation.[1]

The tables in figure 2 specify the strict positions in **A**'s operational statements and expressions. The left-most column indicates with indexed bullets which positions are strict and in what order

[1] Following Cardelli [3] and Reynolds [12], we think of "[t]ype structure [as] a *syntactic* discipline for enforcing levels of abstraction," [12,page 513; emphasis added]. Hence, we qualify all collections of values that arise at *run-time* as run-time types, and do not consider them types *per se*.

Evaluation:

$$
\begin{array}{lcll}
eval_{safe}(P(i)) & = & v & \text{iff } (arg \ = \ i; P) \mapsto^*_{p,s} D;\ \mathsf{skip};\ \mathsf{return}\ v \\
eval_{safe}(P(i)) & = & \mathsf{err}_a & \text{iff } (arg \ = \ i; P) \mapsto^*_{p,s} \mathsf{err}_a \\
eval_{safe}(P(i)) & \Uparrow & & \text{iff } (arg \ = \ i; P) \mapsto^*_{p,s} P' \text{ implies } P' \mapsto_{p,s} \text{ for some } P'
\end{array}
$$

Errors: $\mathbf{Err} = \{\mathsf{err}_{ob}, \mathsf{err}_{if}, \mathsf{err}\cdot/0, \mathsf{err}_l, \mathsf{err}_r\}$

***Safe* Transitions for Erroneous Programs:**

bad array assignment:
$\mathcal{D}\langle x[k] \ = \ \ldots \ u_l \ \ldots\rangle;$ $\mathcal{S}\langle x[w] \ := \ v\rangle;$ $\mapsto_s$ err_{ob}
if $w \notin \mathbf{Z}$ or if $w \in \mathbf{Z}$ but $w \notin [1, \ldots, k]$

bad array dereferenc:
$\mathcal{D}\langle x[k] \ = \ \ldots \ u_l \ \ldots\rangle;$ $\mathcal{E}\langle x[w]\rangle;$ $\mapsto_s$ err_{ob}
if $w \notin \mathbf{Z}$ or if $w \in \mathbf{Z}$ but $w \notin [1, \ldots, k]$

if-not-a-bool:
$D;$ $\mathcal{S}\langle \mathsf{if}\ (k)\ S\ \mathsf{else}\ T\rangle;$ $\mapsto_s$ err_{if}

division-by-zero:
$D;$ $\mathcal{E}\langle k/0\rangle;$ $\mapsto_s$ $\mathsf{err}_{/0}$

not-a-number left-of binary operator:
$D;$ $\mathcal{E}\langle b \otimes v\rangle;$ $\mapsto_s$ err_l

not-a-number right-of binary operator:
$D;$ $\mathcal{E}\langle k \otimes b\rangle;$ $\mapsto_s$ err_r
with $b \notin \mathbf{Z}$ and $k \notin \mathbf{B}$

Figure 4: Safe Completion for **A**

Evaluation:

$$
\begin{array}{lcll}
eval_{unsafe}(P(i)) & = & v & \text{iff } (arg \ = \ i; P) \mapsto^*_{p,u} D;\ \mathsf{skip};\ \mathsf{return}\ v \\
eval_{unsafe}(P(i)) & \Uparrow & & \text{iff } (arg \ = \ i; P) \mapsto^*_{p,u} P' \text{ implies } P' \mapsto_{p,u} \text{ for some } P'
\end{array}
$$

***Unsafe* Transitions for Erroneous Programs:**

bad array assignment:
$\mathcal{D}\langle x[k] \ = \ \ldots; y = v\rangle;$ $\mathcal{S}\langle x[w] \ := \ u\rangle;$ $\mapsto_u$ $\mathcal{D}\langle x[k] \ = \ \ldots; y = u\rangle;$ $\mathcal{S}\langle \mathsf{skip}\rangle;$
if $w \notin \mathbf{Z}$ or if $w \in \mathbf{Z}$ but $w \notin [1, k]$

bad array dereference:
$\mathcal{D}\langle x[k] \ = \ \ldots; y = v\rangle;$ $\mathcal{E}\langle x[w]\rangle;$ $\mapsto_u$ $\mathcal{D}\langle x[k] \ = \ \ldots; y = v\rangle;$ $\mathcal{E}\langle v\rangle;$
if $w \notin \mathbf{Z}$ or if $w \in \mathbf{Z}$ but $w \notin [1, k]$

if-not-a-bool:
$D;$ $\mathcal{S}\langle \mathsf{if}\ (k)\ S\ \mathsf{else}\ T\rangle;$ $\mapsto_u$ $D;$ $\mathcal{S}\langle T\rangle;$

division-by-zero:
$D;$ $\mathcal{E}\langle k/0\rangle;$ $\mapsto_u$ $D;$ $\mathcal{E}\langle 0\rangle;$

not-a-number left-of binary operator:
$D;$ $\mathcal{E}\langle b \otimes v\rangle;$ $\mapsto_u$ $D;$ $\mathcal{E}\langle 1\rangle;$

not-a-number right-of binary operator:
$D;$ $\mathcal{E}\langle k \otimes b\rangle;$ $\mapsto_u$ $D;$ $\mathcal{E}\langle 2\rangle;$
with $b \notin \mathbf{Z}$ and $k \notin \mathbf{B}$

Figure 5: Unsafe Completion for **A**

they are evaluated. The second and third columns then specify the run-time types as subsets of the matching set of values (V for expression values, $\mathbf{B}$ for booleans, $\mathbf{Z}$ for integers, $[1, k]$ for some context-dependent integer interval).

Finally, figure 3 specifies the "instructions" or state transitions of our abstract machine. The transitions are specified as functions that map programs to programs, very much along the lines of the standard reduction of the (imperative) λ_v-CS-calculus [6, 7]. Each transition function in figure 3 describes how a program with a particular body is transformed into some other program and how this transformation may affect the program definitions. The statement rewriting reflects the flow of control; the definitions represent the state of the memory and thus, the effects on the definitions capture the changes in memory.

More technically, the specificiation of each transition function in figure 3 assumes that the program's body has been partitioned into an evaluation context ($\mathcal{S}\langle\bullet\rangle$ or $\mathcal{E}\langle\bullet\rangle$) and a program operation. The grammar for contexts guarantees a unique partitioning. The context is a statement with a single hole, which stands for a statement or an expression; it represents the continuation. The operation in the hole of a context is the next operation to be executed. If it is an assignment and a dereferencing operation, the left-hand side of the transition function also shows the current content of the assigned or referenced variable or array. The right-hand side of each transition function specifies the successor state through new phrases in the holes of the contexts.

Safety: Figure 3 only specifies transitions for programs in which partial operations are applied to proper values. Put differently, if an evaluation reaches a state in one of the following six classes:

1. $\mathcal{D}\langle x[k] \;=\; \ldots; y = v\rangle;\ \mathcal{S}\langle x[w] \;:=\; u\rangle;$ if $w \notin \mathbf{Z}$ or if $w \in \mathbf{Z}$ but $w \notin [1, \ldots, k]$
2. $\mathcal{D}\langle x[k] \;=\; \ldots; y = v\rangle;\ \mathcal{E}\langle x[w]\rangle;$ if $w \notin \mathbf{Z}$ or if $w \in \mathbf{Z}$ but $w \notin [1, \ldots, k]$
3. $D;\ \mathcal{S}\langle \mathsf{if}\ (k)\ S\ \mathsf{else}\ T\rangle;$ if $k \notin \mathbf{B}$
4. $D;\ \mathcal{E}\langle k/0\rangle;$
5. $D;\ \mathcal{E}\langle b \otimes v\rangle;$ if $b \notin \mathbf{Z}$
6. $D;\ \mathcal{E}\langle k \otimes b\rangle;$ if $b \notin \mathbf{Z}$

it is stuck with respect to the reduction function $\mapsto_p$. It is straightforward to verify that these six classes of stuck states and the ten classes of left-hand sides in figure 3 form a partition of the set of all well-formed programs.

A *safe programming language* specifies not only the behavior of program operations on proper but also on improper values. More precisely, a safe language specifies (1) some partial operations, (2) the run-time types of all partial operations as recursive subsets of the set of (possible) values, and (3) what error signal a partial operation raises when it is misapplied. By specifying the precise behavior of the primitive operations, a programming language designer expresses an intent to protect programmers from certain basic mistakes or, put positively, to guarantee some minimal data invariants. Of course, subtle differences in the specifications strongly affect the precise nature of the guarantees.

In contrast, an *unsafe language definition* fails to specify the precise run-time types of operations or their error behavior. Since an *unsafe implementation* typically implements a language's data as bit strings and operations as hardware instructions on bit strings, it is common that all operations become total—modulo safety checks in the underlying operating systems and hardware, which we ignore here.[2] For example, the specification of C—a classical unsafe language—suggests that "it is illegal to refer to objects that are not within the array bounds" [K&R:p. 100], yet, by equating arrays

[2] A more realistic model could introduce errors that reflect such concepts as segmentation faults and bus errors, but this would not affect the conclusions of our work.

and pointers, effectively forces a C implementation to interpret *all* inputs to array dereferencing and assignment.

We model safe and unsafe languages with an extension of the framework of figure 3. For the former, we must specify state transitions for each class of stuck states so that the successor state is an exceptional and final state. To this end, we introduce a class of error states that are also final states: see figure 4. The union of the error transitions in figure 4 and the proper transitions in figure 3 specifies the relation $eval_{safe}$ between programs, inputs ($\mathbf{Z}$), and final answers. It is straightforward to prove a uniform evaluation theorem for $eval_{safe}$.

Theorem 2.1 (Safe Evaluation) *$eval_{safe}$ is a partial function. If P is an $\mathbf{A}$-program and $i \in \mathbf{Z}$, then one of three cases holds exclusively:*

1. $eval_{safe}(P(i)) = k$,
2. $eval_{safe}(P(i)) \in \mathbf{Err}$, *or*
3. $eval_{safe}(P(i)) \Uparrow$.

That is, the safe evaluation of a program either reaches a final state or is infinitely long, but it cannot get stuck. Hence, a program implements a partial function from $\mathbf{Z}$ to $V \cup \mathbf{Err}$.

We model unsafe languages by specifying transition functions between stuck states and successor states from the set of programs. To model the arbitrariness of unsafe data interpretations, we chose some strategy to pick a successor state deterministically, which reflects the traditional reliance on bit-pattern interpretation by implementors of unsafe languages. Consider an array assignment whose index is out of bounds. A typical unsafe implementation writes the value into some other location; the transition function **bad array assignment** in figure 5 approximates that by putting the right-hand side value of the assignment into some location y, which is chosen based on our fixed, but unspecified strategy.[3]

The union of the unsafe tranisitions in figure 5 and those of figure 3 is still a deterministic evaluator.

Theorem 2.2 (Unsafe Evaluation) *$eval_{unsafe}$ is a partial function. If P is an $\mathbf{A}$-program and $i \in \mathbf{Z}$, then one of two cases holds exclusively:*

1. $eval_{unsafe}(P(i)) = k$ *or*
2. $eval_{unsafe}(P(i)) \Uparrow$.

In other words, just like a safe evaluator, the unsafe machine cannot get stuck either. A program interpreted in an unsafe manner is a partial function from $\mathbf{Z}$ to V, *i.e.*, it either terminates with a value (number or boolean) or runs forever.

Consider the following two sample programs:

Example 1:
$x[3] = 1\ 2\ 3;$
$i = \boxed{3};$
$s = 0;$
while $(i \geq 1)$ do {
  $s := s + x[i];$
  $i := i - 1\};$
return s

Example 2:
$x[3] = 1\ 2\ 3;$
$i = \boxed{4};$
$s = 0;$
while $(i \geq 1)$ do {
  $s := s + x[i];$
  $i := i - 1\};$
return s

[3] Our strategy for picking successor states is simplistic and under-specified. The choice of strategy does not affect our results.

The program on the left adds the numbers in the array x and yields 6 on both evaluators. The program on the right, however, reduces to err_{ob} on the safe machine and, if the identifier definition $i = 4$ is the chosen location for $x[4]$, to 10 on the unsafe one.

Here is a general theorem on the relationship of the safe and unsafe evaluators.

Theorem 2.3 *Let P is an* **A***-program and $i \in \mathbf{Z}$.*

1. *If $eval_{safe}(P(i)) = k$, then $eval_{unsafe}(P(i)) = k$;*
2. *If $eval_{safe}(P(i)) \Uparrow$, then $eval_{unsafe}(P(i)) \Uparrow$.*

The converses don't hold. There exist **A***-programs P and Q such that*

1. *$eval_{unsafe}(P(i)) = k$ and $eval_{safe}(P(i)) \in \mathbf{Err}$;*
2. *$eval_{unsafe}(Q(i)) \Uparrow$ and $eval_{safe}(Q(i)) \in \mathbf{Err}$.*

Simply put, the theorem says that an **A**-program that works properly has the same behavior on both evaluators. Unfortunately, for a program that misinterprets data we cannot predict how it behaves under the unsafe evaluator. It could produce a final value or equally well go into an infinite loop. Worse, when a program returns a value on the unsafe evaluator, a programmer cannot be sure that the value is reproducible on a safe evaluator.

In summary, the theorems and examples illustrate the key difference between safe and unsafe languages. A safe language does not produce fewer but more error messages than an unsafe language. Paraphrasing Milner's [10] famous slogan concerning well-typed programs, we could say[4]

safe programs can signal errors; unsafe programs never signal errors.

And the very fact that safe languages do signal errors is their key advantage; the problem of unsafe languages is that every result, no matter how it was computed, appears to be a good result.

Variable Assignment: $\dfrac{\Gamma(x) = t \quad \Gamma \vdash E : t}{\Gamma \vdash_s x := E}$ **Array Assignment:** $\dfrac{\Gamma(x) = t \quad \Gamma \vdash E : t, \quad \Gamma \vdash E_0 : \mathsf{int}}{\Gamma \vdash_s x[E_0] := E}$

If Statements: $\dfrac{\Gamma \vdash E : \mathsf{bool} \quad \Gamma \vdash_s S, \quad \Gamma \vdash_s T}{\Gamma \vdash \mathsf{if}\ (E)\ S\ \mathsf{else}\ T}$ **While Loops:** $\dfrac{\Gamma \vdash E : \mathsf{bool} \quad \Gamma \vdash_s S}{\Gamma \vdash \mathsf{while}\ (E)\ \mathsf{do}\ S}$

Arithmetical Operation: $\dfrac{\Gamma \vdash E_l : \mathsf{int} \quad \Gamma \vdash E_r : \mathsf{int}}{\Gamma \vdash E_l \otimes E_r : \mathsf{int}}$ **Relational Operation:** $\dfrac{\Gamma \vdash E_l : \mathsf{int} \quad \Gamma \vdash E_r : \mathsf{int}}{\Gamma \vdash E_l \oplus E_r : \mathsf{bool}}$

$\Gamma : I \to \mathit{Type} \qquad \Gamma(\mathit{arg}) = \mathsf{int}$

Figure 6: The Type System for **TypedA**

---

[4]Indeed, if a program in an unsafe language triggers an error, it is only due to the safety checks in an operating system or the underlying hardware.

3 From Safety to Types

Like safety checks, type systems are intended to prevent misapplications of primitive operations.[5] Roughly speaking, a type system provides a language of types and a specification of when programs are well-typed. An implementation enforces the well-typing constraints with a type checker and only executes well-typed programs.

Ideally, well-typing implies some minimal semantic invariants, that is, logical statements about the program's evaluation. For example, a programmer who declares a type for some identifier may expect that this identifier always stands for run-time values of that type. Similarly, if an expression has a certain type, the programmer should expect that the expression always yields values of that type (or diverges). This relationship has become known as Milner's type soundness theorem [10]; it is often translated into the slogan that typed programs can't go wrong, *i.e.*, can't misapply operations. Unfortunately, this statement only holds in highly simplified languages like **A** without arrays and division. It does not hold for a typed version of **A** as we will show in this section.

The language **TypedA** is a typed variant of **A**. In **TypedA**, definitions must specify the types of identifiers and of arrays:

$$\begin{array}{rcl} P & = & Dt;\ S;\ \mathsf{return}\ I \\ Dt & = & dt; \ldots;\ dt \\ dt & = & \mathit{Type}\ I\ =\ V \mid \mathit{Type}\ I[V]\ =\ V \ldots V \\ \\ \mathit{Type} & = & \mathsf{int} \mid \mathsf{bool} \end{array}$$

The rest of the syntax is the same as for **A**. The type constraints are specified via the simple logic in figure 6. The logic's rules are close to those for typed languages like C or ML. They specify when, in some type context Γ, which is a function from identifiers to types, an expression has type int or bool and when a statement is valid.

Otherwise, **TypedA** and **A** are basically the same. In particular, the two languages have the same primitive program operations with the same run-time types. Adapting the safe and the unsafe evaluators to the typed syntax is straightforward. We continue to refer to the two respective evaluation functions as $\mathit{eval}_{\mathit{safe}}$ and $\mathit{eval}_{\mathit{unsafe}}$.

The type system of **TypedA** prevents the programmer from making certain mistakes. For example, the following program

$$\mathsf{int}\ x\ =\ 0; \ldots \mathsf{if}\ (x)\ x\ :=\ 1\ \mathsf{else}\ x\ :=\ 0$$

would not pass the type checker and an evaluator of **TypedA** would therefore reject to evaluate the program. On the safe machine for **A**, the same program would trigger a run-time error, which is why the type system is designed to judge it as "bad." More generally, the type system for **TypedA** reduces the set of possible run-time errors on the *safe* evaluator that a program may signal.

Theorem 3.1 (Type Soundness) *If P is an* **TypedA***-program and $i \in \mathbf{Z}$, then one of three cases holds exclusively*

1. $\mathit{eval}_{\mathit{safe}}(P(i)) = k$,
2. $\mathit{eval}_{\mathit{safe}}(P(i)) \in \{\mathsf{err}_{ob}, \mathsf{err}_{/0}\}$, *or*
3. $\mathit{eval}_{\mathit{safe}}(P(i)) \Uparrow$.

[5] A type system can also filter out other programs that are undesirable, for example, because they are extremely expensive to implement.

This theorem is the correct generalization of Milner's [10] type soundness theorem to realistic languages. It does not claim that typed programs do not produce errors, but that *typed programs on safe languages have fewer ways to signal errors than equivalent untyped programs.*

The type system[6] does not make the language safe. The proof of the above theorem shows that a **TypedA** program can reach three of kinds of stuck states: array assignment out of bounds, array dereferencing out of bounds, and division by 0. Hence, an unsafe interpretation of such faulty programs can still misapply operations and thus misinterpret data.

Theorem 3.2 (Type Unsoundness) *For each of the following three classes of stuck states:*

1. $\mathcal{D}\langle x[k] \ = \ \ldots; y = v\rangle;\ \mathcal{S}\langle x[w] \ := \ u\rangle;$ *if* $w \notin \mathbf{Z}$ *or if* $w \in \mathbf{Z}$ *but* $w \notin [1, \ldots, k]$,
2. $\mathcal{D}\langle x[k] \ = \ \ldots; y = v\rangle;\ \mathcal{E}\langle x[w]\rangle;$ *if* $w \notin \mathbf{Z}$ *or if* $w \in \mathbf{Z}$ *but* $w \notin [1, \ldots, k]$, *and*
3. $D;\ \mathcal{E}\langle v/0\rangle;$,

there exists a **TypedA**-*program* P *and* $i \in \mathbf{Z}$ *such that*

$$P(i) \mapsto^*_{p,u} Q$$

and Q *is an element of the chosen class.*

Note: Assuming the error transitions satisfy certain simple criteria, all stuck states are reachable.

Example 2 is a simplistic example for this theorem. After declaring each identifier and array to be of type int, it type-checks yet still misinterprets array x. Using this form of out-of-bounds indexing, a **TypedA** program can store any value into an identifier of any type. In short, a typed program can misapply primitive operations on an unsafe evaluator.

4 Safety and Dynamic Memory Usage

An **A** program allocates a fixed amount of memory at the beginning of an evaluation and does not allocate any further memory. Practical languages allow programs to perform functional calls and to allocate memory from a heap. Both actions increase the amount of memory during evaluation.

The final version of this section will contain precise theorems.

Adding memory-consuming primitives to **A** poses a new challenge for the formulation of safety. To illustrate this challenge, we develop **PairA**, an extension of **A** that includes the pair constructor pair and selectors first and second. Figure 7 specifies the syntax and two abstract machines for **PairA**. The extended expression syntax includes three new expressions. We also add a finite number of locations, which are needed to model allocation. Locations are values and always stand for dynamically allocated pairs.

The operations pair, first, and second are value strict in the expected manner. To model their behavior, we add a heap to the states of our abstract machine. A heap is a partial, finite function from the set of locations to expressions of the form $\mathsf{pair}(V, V)$. If a heap is defined at location l, we write

$$H[l = \mathsf{pair}(v, u)]\ ;$$

otherwise, we write

$$H[l]\ .$$

All machine transitions in the core machine (figures 3, 4 and 5) are augmented with a heap whose presence has no effect.

The core transitions of **PairA** are as expected. A pair-expression allocates an unused location; first and second access the appropriate fields of a pair in the heap. The addition of locations as

[6]Indeed, *no* type system turns an unsafe version of **A** into a safe language, assuming the language is equivalent to Turing machines and has operations like division or array indexing.

Syntax:

$$\begin{array}{rcl} E & = & \ldots \mid \mathsf{pair}(E,E) \mid \mathsf{first}(E) \mid \mathsf{second}(E) \mid \mathbf{L} \\ \mathbf{L} & = & l_1 \mid \ldots \mid l_n \\ V & = & \ldots \mid \mathbf{L} \end{array}$$

Strictness:

Expressions		
$\mathsf{pair}(\bullet_1, \bullet_2)$	V	V
$\mathsf{first}(\bullet_1)$	$\mathbf{L}$	
$\mathsf{second}(\bullet_1)$	$\mathbf{L}$	

Context: $\mathcal{F}\langle\bullet\rangle = \ldots \mathsf{pair}(\mathcal{F}\langle\bullet\rangle, E) \mid \mathsf{pair}(V, \mathcal{F}\langle\bullet\rangle) \mid \mathsf{first}(\mathcal{F}\langle\bullet\rangle) \mid \mathsf{second}(\mathcal{F}\langle\bullet\rangle)$

Core Transitions:

pair:
$H[l]; \quad D; \quad \mathcal{E}\langle\mathsf{pair}(v,u)\rangle; \quad \mapsto_m \quad H[l = \mathsf{pair}(v,u)]; \quad D; \quad \mathcal{E}\langle l\rangle;$

first:
$H[l = \mathsf{pair}(v,u)]; \quad D; \quad \mathcal{E}\langle\mathsf{first}(l)\rangle; \quad \mapsto_m \quad H[l = \mathsf{pair}(v,u)]; \quad D; \quad \mathcal{E}\langle v\rangle;$

second:
$H[l = \mathsf{pair}(v,u)]; \quad D; \quad \mathcal{E}\langle\mathsf{second}(l)\rangle; \quad \mapsto_m \quad H[l = \mathsf{pair}(v,u)]; \quad D; \quad \mathcal{E}\langle u\rangle;$

Safe **Transitions**:

out of memory:
$H; \quad D; \quad \mathcal{E}\langle\mathsf{pair}(v,u)\rangle; \quad \mapsto_m \quad \mathsf{err}_{oom}$
if $dom(H) = \mathbf{L}$

first on unused location:
$H[l]; \quad D; \quad \mathcal{E}\langle\mathsf{first}(l)\rangle; \quad \mapsto_m \quad \mathsf{err}_{first,u}$

first on non-location:
$H; \quad D; \quad \mathcal{E}\langle\mathsf{first}(v)\rangle; \quad \mapsto_m \quad \mathsf{err}_{first,v}$
if $v \notin \mathbf{L}$

second on unused location:
$H[l]; \quad D; \quad \mathcal{E}\langle\mathsf{second}(l)\rangle; \quad \mapsto_m \quad \mathsf{err}_{second,u}$

second on non-location:
$H; \quad D; \quad \mathcal{E}\langle\mathsf{second}(v)\rangle; \quad \mapsto_m \quad \mathsf{err}_{second,v}$
if $v \notin \mathbf{L}$

Unsafe **Transitions**:

out of memory:
$H; \quad D; \quad \mathcal{E}\langle\mathsf{pair}(v,u)\rangle; \quad \mapsto_m \quad H; \quad D; \quad \mathcal{E}\langle 0\rangle;$
if $dom(H) = \mathbf{L}$

first on unused location:
$H[l]; \quad D; \quad \mathcal{E}\langle\mathsf{first}(l)\rangle; \quad \mapsto_m \quad H; \quad D; \quad \mathcal{E}\langle 0\rangle;$

first on non-location:
$H; \quad D; \quad \mathcal{E}\langle\mathsf{first}(v)\rangle; \quad \mapsto_m \quad H; \quad D; \quad \mathcal{E}\langle 0\rangle;$
if $v \notin \mathbf{L}$

second on unused location:
$H[l]; \quad D; \quad \mathcal{E}\langle\mathsf{second}(l)\rangle; \quad \mapsto_m \quad H; \quad D; \quad \mathcal{E}\langle 0\rangle;$

second on non-location:
$H; \quad D; \quad \mathcal{E}\langle\mathsf{second}(v)\rangle; \quad \mapsto_m \quad H; \quad D; \quad \mathcal{E}\langle 0\rangle;$
if $v \notin \mathbf{L}$

Figure 7: Safe and Unsafe Models of **ConsA**

values induces new stuck states with respect to the core semantics ($\mapsto_p$). Specifically, a program may erroneously apply first (and second) to a non-location or an **A**-operation to a location. In addition, first (and second) could be applied to a location without contents. All of these problems are analogous to those encountered in section 2 and are dealt with appropriately.

The finiteness of **L** introduces a new kind of stuck state. If the domain of the heap H is the entire set of locations, *i.e.*, no location is unused, then

$$H; D;\ \mathcal{E}\langle \mathsf{pair}(v,u)\rangle;\ \mathsf{return}\ x$$

is a stuck state.[7] A safe version of **PairA** should signal an error when it encounters this state so that the programmer (or user) knows that the program ran out of memory. An unsafe version may continue the evaluation with 0 or some other non-sensical location instead.

Figure 7 specifies the safe and unsafe completions of the core semantics. The safe transitions abort the program evaluation. The unsafe transitions always produce 0 as non-sensical values for mis-applications of primitives.

The model of **PairA** correctly captures memory allocation, but fails to support reclamation of memory, a standard operation in dynamic languages. Such languages provide either automatic memory management, usually through garbage collection, or a construct for deallocating memory locations explicitly. Modeling garbage collection in a reduction semantics is straightforward [7]:

garbage collection:

$$H \uplus H';\quad D;\quad S;\ \mathsf{return}\ x \quad \mapsto_m \quad H \uplus dom(H');\quad D;\quad S;\ \mathsf{return}\ x$$
$$\text{if } dom(H') \cap (\mathbf{L}(rng(H)) \cup \mathbf{L}(D;S)) = \emptyset$$

In other words, an evaluator may discard the pair-cells in all those locations that do not occur in the rest of the heap or the program.

The left-hand side of the transition overlaps with those of the other transitions and therefore introduces a degree of indeterminism into our abstract machine. Still, we can show that the Safe Evaluation theorem of section 2 holds for the extended evaluator. Moreover, we can adapt Morrisett et al.'s [11] techniques to prove that the extended evaluator is *sound*. That is, using the new rules does not affect the observable answers—except that the addition of a garbage collector in a *finite* memory setting may *eliminate* errors. This, of course, implies an important **observation**:

garbage collection reduces the **likelihood** *of run-time errors.*

Specifically, garbage collection makes it more likely that an application of pair succeeds.

Here is an extension of **PairA** with a free-statement, that is, a programmer-controlled construct for deallocating memory:

free:

$$H[l = \mathsf{pair}(u,v)];\quad D;\quad \mathcal{S}\langle \mathsf{free}(l)\rangle;\ \mathsf{return}\ x \quad \mapsto_m \quad H[l];\quad D;\quad \mathcal{S}\langle \mathsf{skip}\rangle;\ \mathsf{return}\ x$$

free unallocated:

$$H[l];\quad D;\quad \mathcal{S}\langle \mathsf{free}(l)\rangle;\ \mathsf{return}\ x \quad \mapsto_m \quad H[l];\quad D;\quad \mathcal{S}\langle \mathsf{skip}\rangle;\ \mathsf{return}\ x$$

The rules specify that a location is made available for future allocations, regardless of whether it is currently in use. As we have long recognized, these rules are *unsound*. It creates dangling pointer, which, in turn, may affect the observable outputs of the program. In the case of **PairA**, a program that used to return one number may return a different one now. To overcome this lack of soundness, we must add the following precondition to the first rule:

$$l \notin (\mathbf{L}(rng(H)) \cup \mathbf{L}(D;\ S;\ \mathsf{return}\ x))$$

[7] Put differently, we can think of the heap as a third argument to pair, and for a fully used heap, pair is not defined.

This precondition is effectively equivalent to that of the garbage collection rule and makes the free-operation at least as expensive.

Surprisingly though, our current definition of safety does *not* say that the manual deallocation rules corrupt safety. One could argue that the operation is not partial on allocated locations and that it always succeeds. Furthermore, the collection of heap values has a coherent interpretation—as long as interpretation stops at deallocated locations. And, if a program eventually uses first or second on such a deallocated location, the safe language aborts the evaluation. We believe that this gray area is acceptable; after all, it is up to the programmer to decided whether the safety guarantees of a language suffice for some projects.

In summary, while garbage collection is not indispensable for the safe evaluation of a dynamic language, it is the only way to make memory management sound. Furthermore, garbage collection is as important as a type system for reducing the number of potential run-time errors. In contrast, a free-statement in a typed language not only corrupts the soundness of the evaluator; it also undermines the soundness of the type system and thus the primary purpose of the type system.

The final paper will include an example that illustrates this problem with type soundness.

5 Related Work

The literature on programming languages does not contain a rigorous definition of safety. Hoare [9] was one of the first to recognize the advantages of safe languages (calling them "secure"), though he did not offer a definition. In his study of type systems, Cardelli [3] comes close to our definition of safety. He informally describes safety as a property of execution, but also suggests that "typed languages may enforce safety by statically rejecting all programs that are potentially unsafe." Based on our investigation, this can only hold if the language is overly simple (or if the type checker may diverge). In general, types according to Cardelli's view encode a notion of good program behavior, different from, but related to, safety. Since Cardelli does not develop a formal semantics, he does not make formal or rigorous claims about the relationship between types and safety.

Alpern and Schneider [1] have characterized properties of (concurrent) evaluations as safety and liveness properties. According to them, a property is a safety property if a uniquely identifiable, first point in the evaluation ensures that the evaluation satisfies the property. Clearly, the property of "misapplying a primitive" is a safety property in this sense. In contrast, a property is a liveness property if a sequence of events of one kind is guaranteed to be followed by an event of a second kind. Hence, collecting unobservable memory is a liveness property. A sequence of allocation events is eventually followed by a memory reclamation event (or a termination event). While Alpern and Schneider's work characterizes these ideas in general terms, it does not define "safe language"—after all both a safe and an unsafe language satisfy one of their safety properties.

6 Conclusions and Future Work

This paper is the first to spell out a rigorous definition of the folklore notion of safety. Roughly speaking, safety encodes some minimal guarantees concerning the coherence of data during execution. The same syntax can be equipped with different, more or less stringent, notions of safety. It is for the programmer to decide whether the guarantees that the language designer (implementor) provides are interesting and helpful for software engineering.

The study of **A** and its variants also reveals a misunderstanding about the relationship between safety, errors, and type systems. The problem is that (except for hardware and OS limitations) unsafe programs *don't* signal errors; safe programs do, i.e., they notify users when the integrity of data (as defined by the language definer) would have been violated if the execution had continued.

Type systems only reduce the number of potential problems in a safe program. More generally, a program analysis is only meaningful if the target language is safe. Conversely, analyzing programs

in an unsafe language is *meaningless*.[8] If such an analyzed program contains a *single* potential fault site, it may still control life-critical devices based on entirely nonsensical intermediate results.

Our definition of safety is a starting point for the development of semantics-based software "metrics." While traditional type systems can eliminate entire classes of safety checks, a safety-based analyzer will inspect each individual operation in a program. It will try to validate that the operation is not misapplied; if it can't, it will flag the operation and explain to the programmer how a bad value may flow to this site. The programmer can then decide whether the analysis's logic is too weak to prove the soundness or whether the example reveals a bug. Bourdoncle's [2] static debugger for Pascal and Flanagan et al.'s [8] soft typer for Scheme are first steps in this direction. If we wish to put software engineering on a semantic foundation, we should investigate further possibilities along these directions.

References

1. Alpern, B. and F. B. Schneider. Defining liveness. *Information Processing Letters* **21** (1985), 181–185.
2. Bourdoncle, F. Abstract debugging of higher-order imperative languages. In *Proc. Sigplan '93 Conference on Programming Language Design and Implementation*, 1993, 44-55.
3. Cardelli, L. Type systems. In *Handbook on Computer Science and Engineering.* Chapter 103. CRC Press. 1997.
4. Cardelli, L. and P. Wegner. On understanding types, data abstractions, and polymorphism. *Computing Surveys* **17**(4), 1985, 471–522.
5. Cristian, F. Correct and robust programs. *IEEE Trans. on Software Engineering, Vol.* **10(2)**. March 1984, 163–174.
6. Felleisen, M. and D.P. Friedman. A syntactic theory of sequential state. *Theor. Comput. Sci.* **69**(3), 1989, 243–287. Preliminary version in: *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987, 314-325.
7. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989. *Theor. Comput. Sci.* **102**, 1992, 235–271.
8. Flanagan, C., M. Flatt, Shriram K., S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proc. Sigplan '96 Conference on Programming Language Design and Implementation*, 1996, 22–32.
9. Hoare, C.A.R. Hints on programming language design. In *Computer Systems Reliability.* C. Bunyan (Ed). Pergamon Press, 1974. 505-534. Reprinted in: *Essays in Computing*, C.A.R. Hoare and C.B. Jones (Eds). Prentice Hall International. Cambridge, 1989.
10. Milner, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **17**, 1978, 348–375.
11. Morrisett, G., M. Felleisen, and R. Harper. Modeling memory management In *Proc. Symposium on Functional Programming and Computer Architectures*, 1995, 66–77.
12. Reynolds, J.C. Types, abstraction and parametric polymorphism. In *Proceedings of the IFIP 9th World Computer Congress.* Elsevier Science Publishers B. V. (North-Holland). Amsterdam, 1983.
13. Wright, A. and M. Felleisen. A syntactic approach to type soundness. Technical Report 160. Rice University, 1991. *Information and Computation* **115**(1), 1994, 38–94.

[8] A program analysis for an unsafe language is still *useful* because it reduces the number of candidates for a *first* fault. But programmers and researchers must understand that an analysis implies nothing about the integrity of data.