Process Migration and Transactions
Using a Novel Intermediate Language\textsuperscript{1}


Jason Hickey      Justin D. Smith      Brian Aydemir
Nathaniel Gray    Adam Granicz         Cristian Tapus

November 16, 2002

\textsuperscript{1}This technical report is based on version 1.76 of the FIR.
## Contents

1 Introduction .................................................. 1  
   1.1 Related work ............................................. 2  
   1.2 System architecture ....................................... 3  

2 The FIR Syntax ............................................... 5  
   2.1 Basic syntax and notation ................................... 5  
   2.2 FIR type system ........................................... 6  
      2.2.1 Basic types .......................................... 6  
      2.2.2 Type definitions ..................................... 7  
   2.3 FIR expressions ........................................... 8  
      2.3.1 Atoms ................................................ 9  
      2.3.2 Expressions ......................................... 11  

3 Judgments .................................................... 15  
   3.1 Heap and store values ..................................... 15  
   3.2 Kinds ..................................................... 16  
   3.3 Contexts .................................................. 16  
   3.4 Judgments ................................................ 17  
      3.5 Free variables and substitution ......................... 17  
         3.5.1 Domain of context $\Gamma$ .......................... 18  
         3.5.2 Free variables ...................................... 18  
         3.5.3 Type substitution .................................. 21  

4 Typing Rules ................................................ 23  
   4.1 Context well-formedness ................................... 23  
   4.2 Type equality and structural rules ......................... 24  
   4.3 Application and elimination type equality rules ........... 24  
   4.4 Constructor equality ..................................... 25  
   4.5 Store typing rules ....................................... 26  
   4.6 Type equality ............................................. 27  
   4.7 Atom typing rules ........................................ 28  
   4.8 Expression typing ........................................ 30  
      4.8.1 Basic expression typing rules ....................... 30  
      4.8.2 Special-call typing rules ........................... 31  
      4.8.3 Match statement typing rules ......................... 32  
      4.8.4 Allocation typing rules .............................. 33  
      4.8.5 Subscripting typing rules ............................ 34  
      4.8.6 Global label typing rules ............................ 35  

5 Operational Semantics ...................................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Evaluation contexts</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Atom evaluation</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Basic expressions</td>
<td>40</td>
</tr>
<tr>
<td>5.4</td>
<td>Special-calls</td>
<td>40</td>
</tr>
<tr>
<td>5.5</td>
<td>Allocation expressions</td>
<td>41</td>
</tr>
<tr>
<td>5.6</td>
<td>Match statements</td>
<td>42</td>
</tr>
<tr>
<td>5.7</td>
<td>Subscripting operations</td>
<td>42</td>
</tr>
<tr>
<td>5.8</td>
<td>Global variable expressions</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Basic FIR properties</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Well-formedness properties</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Proof induction</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Type uniqueness</td>
<td>52</td>
</tr>
<tr>
<td>6.4</td>
<td>Substitution</td>
<td>54</td>
</tr>
<tr>
<td>6.5</td>
<td>Type substitution</td>
<td>69</td>
</tr>
<tr>
<td>6.6</td>
<td>Fully-defined contexts</td>
<td>72</td>
</tr>
<tr>
<td>6.7</td>
<td>Progress lemmas</td>
<td>76</td>
</tr>
<tr>
<td>7</td>
<td>Type safety</td>
<td>79</td>
</tr>
<tr>
<td>7.1</td>
<td>Preservation</td>
<td>79</td>
</tr>
<tr>
<td>7.2</td>
<td>Progress</td>
<td>89</td>
</tr>
<tr>
<td>8</td>
<td>Runtime Implementation</td>
<td>97</td>
</tr>
<tr>
<td>8.1</td>
<td>Runtime data structures and invariants</td>
<td>97</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Data blocks and the heap</td>
<td>98</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Pointer table</td>
<td>98</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Function pointers</td>
<td>99</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Pointer safety</td>
<td>100</td>
</tr>
<tr>
<td>8.2</td>
<td>Process migration</td>
<td>100</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Using process migration in the FIR</td>
<td>101</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Runtime support for migration</td>
<td>102</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Uses for process migration</td>
<td>102</td>
</tr>
<tr>
<td>8.3</td>
<td>Atomic operations and transactions</td>
<td>103</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Using atomic transactions in the FIR</td>
<td>103</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Implementation of atomic transactions</td>
<td>104</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Using atomic transactions for traditional transactions</td>
<td>106</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Using atomic transactions for fault-tolerant computing</td>
<td>106</td>
</tr>
<tr>
<td>8.4</td>
<td>An implementation of fault-tolerant distributed computing</td>
<td>106</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Architecture of the Mojave Compiler ........................................... 3

2.1 Basic FIR terms ........................................................................... 6
2.2 The FIR type system ................................................................. 8
2.3 FIR type definitions ................................................................. 8
2.4 FIR atoms ................................................................................. 10
2.5 FIR unary operators ................................................................. 10
2.6 FIR binary comparison operators .............................................. 11
2.7 FIR binary operators ............................................................... 12
2.8 FIR expressions .......................................................... 14
2.9 FIR operators .......................................................... 14

3.1 Heap values .......................................................... 15
3.2 Store values .......................................................... 16
3.3 Kinds .......................................................... 16
3.4 Program contexts ........................................................... 17
3.5 Judgments .......................................................... 18
3.6 Type free variables ......................................................... 19
3.7 Type definition free variables ........................................... 19
3.8 Atom free variables .......................................................... 20
3.9 Expression free variables .................................................. 20
3.10 Allocation operator free variables ........................................ 20
3.11 Special-call free variables .................................................. 21
3.12 Store free variables .......................................................... 21
3.13 Definition free variables .................................................. 21
3.14 Type substitution .......................................................... 22
3.15 Type definition substitution ................................................ 22

5.1 Evaluation contexts ........................................................... 39

6.1 Thinning and substitution rules ............................................... 50

8.1 Pointer table representation .................................................. 99
8.2 Heap data with multiple atomic levels ................................... 105
Chapter 1

Introduction

The design of software for distributed applications is a challenging task. In addition to the difficulties posed by processor and network failures, distributed applications are often composed of several parts written in different languages. In this technical report, we approach these problems by using a typed, semi-functional intermediate language that supports two key features for distributed computing: whole-process migration, and undoable transactions. Among other services, process migration provides location transparency and the ability to perform load-balancing and process checkpointing. Transactions provide fault-isolation by limiting the scope of errors, and speculative execution by providing the ability to rollback overly optimistic computations. To address the multi-language issue, the intermediate language is designed to support both type-safe source languages like ML, and unsafe languages like C.

We have implemented our approach in the Mojave system, which currently provides a multi-language compiler for programs written in C, Naml (a language based on Caml-light), and Pascal. Programs in these languages are compiled to a typed “Functional Intermediate Representation” (FIR). Types are maintained throughout the compilation process, which ensures that component interactions respect type and memory safety. The core FIR language is a variant of System F [7] in continuation-passing style (CPS) [1]. In many respects, the FIR language is similar to the typed intermediate language used in the TILT compiler [18, 15]. However, the FIR is much more limited syntactically; the precise language is specified in Chapter 2.

The FIR language has been quite robust as new source languages have been added to the Mojave compiler, but there are tradeoffs, most evident when compiling C. First, the FIR requires that program execution be safe, despite the fact that it is not always possible to maintain exact type information for C. This limitation is resolved using runtime safety checks. Pointers in the runtime are represented with base/offset pairs, requiring more storage and extra computation. Second, the decision to use continuation-passing style means that there is no runtime stack, and C functions allocate storage on the heap, which can be expensive in some cases. However, there are also many advantages. The FIR has made it easy to augment ANSI C with polymorphism and type inference, exceptions, pattern matching, higher-order functions, and safe return of pointers to “automatic” variables.

The technical report is organized as follows. In Section 1.1, we discuss related work. Section 1.2 covers the system architecture and design of the Mojave compiler. Chapters 2–5 cover the FIR, including the type system and operational semantics. In Chapter 6, we present a set of basic lemmas needed to prove the type-safety theorems of Chapter 7. We conclude with a discussion of the runtime implementation in Chapter 8.
1.1 Related work

The desire for common intermediate languages dates back to the 1950's, significantly with the UNCOL project [13]. More modern versions that support at least part of the goals of multi-language platforms are the Java Virtual Machine [5] and Microsoft's Common Language Runtime [14, 8]. The JVM has many desirable features: it is portable, it is safe, and compiler technology can produce very efficient executables. However, as Meijer points out [14], while the JVM was designed to be generic, it works most effectively for Java. There is little support for (at one extreme) unsafe languages like C, and (at another extreme) functional languages like ML.

The Microsoft CLR addresses many of these problems, and supports a wide range of languages including parts of C++, OCaml, and Standard ML. However, the CLR does not ensure safety, although Gordon and Syme have been able to demonstrate safety for a substantial fragment [8]. The CLR is a fairly large language. In contrast, the FIR is quite small—and consequently more effort is required from a language front-end to compile to FIR.

Another related area is portable code generators, of which there are several, including MLRISC [6], C– [12, 11], and gcc [17]. As Peyton-Jones suggests, C itself should be considered as a commonly-used assembly language. In each of these cases, the code generator is attacking the problem of portability and machine-code generation, but not the problem of safety.

In the area of C program safety, the Necula et.al. CCured compiler [16] and the Morrisett et.al. Cyclone Safe-C [10] compiler both extend the C language to include extra information needed to infer safety. The systems distinguish between “safe” and “unsafe” pointers, where a safe pointer is always used in ways that can be shown to be safe, and unsafe pointers include all the rest. In both systems, unsafe pointers are represented at runtime by a “fat” pointer that includes safety information, requiring twice the storage of a safe pointer.

CCured extends the type system and uses type inference to determine safety. Cyclone requires explicit annotation by the user: safe and unsafe pointers have different types, and different dereference operators. Compilation is limited to the subset of C that can be proven safe.

In contrast with this work, the Mojave compiler accepts all source files that conform to the ANSI standard, but all pointers are represented as fat pointers. We use dead-code elimination coupled with alias analysis to delete provably unnecessary safety information for variables, but this still requires more space than the other two systems. There is no way in Mojave to represent an array of safe pointers; all pointers are 8 bytes on all platforms.

Process migration has been widely studied [20, 4], yet the availability for general-purpose languages, like C and ML, is limited. Our approach to process migration has been heavily influenced by Cardelli’s work on the Ambient Calculus [3, 2]; however, our work with whole-process migration is only the first step toward fine-grained mobility.

Transactions are a fundamental concept in the database community, but again, implementations for general-purpose languages are limited. As part of the Venari project, Haines et.al. [9] implement a transaction mechanism as part of Standard ML. Undoability is implemented by extending the mutation log produced by the generational garbage collector. Our approach (described in Section 8.3.2) also uses a mutation log. However, we combine a generational mark-sweep collector with a copy-on-write mechanism to reduce the cost of rollback and commit operations.
1.2 System architecture

The Mojave system has three major parts. The front-ends are responsible for compiling code to FIR, and the back-ends are responsible for generating machine code from FIR. FIR type inference and optimizations form the middle stage of the compiler. The stages of the compiler are shown in Figure 1.1 for the C and Naml source languages; Pascal is compiled through a preprocessor to the C front-end. Currently, there is one back-end, for the Intel IA32 architecture.

![Architecture Diagram]

Figure 1.1: Architecture of the Mojave Compiler

As shown in the diagram, the front-ends perform a significant amount of compilation to produce the FIR representation. In particular, the C front-end has to add explicit coercions and eliminate pointer arithmetic. In addition, all of the branches perform CPS and closure conversion. We do not discuss the front-ends in detail in this paper, focusing instead on the FIR and runtime implementation.
Chapter 2

The FIR Syntax

The figures below define the syntactically valid terms of the FIR. Each term is accompanied by a brief description, and in most cases, the corresponding FIR “entity”, an OCaml expression or type used to implement that term\(^1\). The precise meanings of these terms are given by the typing rules (Chapter 4) and operational semantics (Chapter 5).

2.1 Basic syntax and notation

In the syntax descriptions below, we use the following conventions. In general, we use the meta-variables \(i, j, \) and \(k\) to refer to arbitrary integers, and the meta-variables \(m\) and \(n\) to refer to arbitrary nonnegative integers. In most cases, we use the meta-variable \(m\) to enumerate type parameters \(\alpha_1, \ldots, \alpha_m\), and the meta-variable \(n\) to enumerate actual parameters \(v_1, \ldots, v_n\). The runtime values are indexed from zero; in cases where this matters, we enumerate type parameters with \(\alpha_0, \ldots, \alpha_{m-1}\) and actual parameters with \(v_0, \ldots, v_{n-1}\).

The meta-variable \(v\) refers to program variables. The meta-variable \(l\) refers to program labels, which are used as record field tags and names for global variables. The labels and variables are distinct sets.

The meta-variables \(t\) and \(u\) refer to program types, while the Greek letters \(\alpha, \beta, \gamma\) and the meta-variable \(tv\) refer to type variables.

We use the notation \([v_i]_1^n\) to denote the vector \(v_1, \ldots, v_n\). The index variable is implicitly \(i\). An alternate notation \([v_j]_{j=1}^n\) explicitly uses a different index variable \(j\).

The basic program syntax is shown in Figure 2.1. There are several forms of numbers. In addition to the integers \(i\), there are raw integers with various bit precisions (8, 16, 32, and 64) and signedness interpretations (signed and unsigned). The floating-point values also have several precisions (32, 64, and 80 bits). The meta-variable \(x\) stands for a floating-point value.

Sets of integers are used in integer “pattern matching” expressions. The sets are represented by lists of closed

\(^1\)These are mainly for the benefit of the authors and we omit any discussion of these implementation details from the paper. * is used in these entities to mark an expression that is used only as a hint for debugging purposes or optimization, but has no effect on the computation.
intervals \([i_1, i_2]\). In the case of raw integer sets, the end points of each interval can have any of the several bit precisions. We also commonly represent a singleton set containing the integer \(i\) using the notation \(\{i\}\).

There are two kinds of tuples, *normal* tuples and *box* tuples. The box tuples always have arity one, and are used to pass arbitrary values (such as large precision integers, or floating-point values) as polymorphic values.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
<th>FIR Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_1, v_2, \ldots)</td>
<td>Variable names</td>
<td>Fir.var</td>
</tr>
<tr>
<td>(tv_1, tv_2, \ldots)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\alpha, \beta, \ldots)</td>
<td>Type variables</td>
<td>Fir.ty.var</td>
</tr>
<tr>
<td>(l_1, l_2, \ldots)</td>
<td>Labels</td>
<td>Fir.label</td>
</tr>
<tr>
<td>(i := \ldots \mid -1 \mid 0 \mid 1 \mid \ldots)</td>
<td>Integer constants</td>
<td>int</td>
</tr>
<tr>
<td>(pre := 8 \mid 16 \mid 32 \mid 64)</td>
<td>Integer precisions</td>
<td>int8</td>
</tr>
<tr>
<td>(sign := \text{signed} \mid \text{unsigned})</td>
<td>Integer signedness</td>
<td>true</td>
</tr>
<tr>
<td>(r := (i, pre, sign))</td>
<td>Raw integer constants</td>
<td>Rawint.rawint</td>
</tr>
<tr>
<td>(fpre := 32 \mid 64 \mid 80)</td>
<td>Floating-point precisions</td>
<td>Single</td>
</tr>
<tr>
<td>(x := \text{float})</td>
<td>Floating-point constants</td>
<td>Rawfloat.rawfloat</td>
</tr>
<tr>
<td>(set_I := [i_1^1, i_2^1], \ldots, [i_1^n, i_2^n])</td>
<td>Integer interval set</td>
<td>IntSet.t</td>
</tr>
<tr>
<td>(set_R := [r_1^1, r_2^1], \ldots, [r_1^n, r_2^n])</td>
<td>Raw integer interval set</td>
<td>RawIntSet.t</td>
</tr>
<tr>
<td>(set := set_I \mid set_R)</td>
<td>Arbitrary sets</td>
<td>IntSet set_I \mid RawIntSet set_R</td>
</tr>
<tr>
<td>(\text{tuple.class} := \text{normal} \mid \text{box})</td>
<td>Tuple classes</td>
<td>NormalTuple</td>
</tr>
</tbody>
</table>

Figure 2.1: Basic FIR terms

2.2 FIR type system

The FIR has two classes of types, the basic types, shown in Figure 2.2, and the type definitions, shown in Figure 2.3. The types are described more fully in Chapter 4, which defines the program typing rules.

2.2.1 Basic types

**Numbers** The notation \(\mathbb{Z}_{31}\) refers to a 31-bit signed integer value. The runtime can always distinguish \(\mathbb{Z}_{31}\) values from pointers\(^2\). The type \(\text{enum}_i\) includes the integers \(\{0, \ldots, i-1\}\). The maximum enumeration value is bounded by the value \(\text{max_enum}^i\). The type \(\text{enum}_0\) is valid and represents the void type. The native integer type \(\mathbb{Z}_{\text{sign}}^{\text{pre}}\) includes values of a specified bit precision and signedness interpretation. The type \(\mathbb{R}_{\text{pre}}\) includes floating-point values of a specified bit precision.

**Tuples** The tuple type \(\langle t_1, \ldots, t_n \rangle_{\text{tuple.class}}\) represents a tuple \(\langle v_1, \ldots, v_n \rangle\), where each value \(v_i\) has type \(t_i\).

\(^2\)To distinguish integers from pointers, the runtime often represents the 31-bit quantity \(i\) as a 32-bit value \(2i + 1\); pointers always have have a least-significant-bit of 0.

\(^3\)This is explained further in a footnote in Section 4.6.
**Arrays** Arrays resemble tuples, but the elements of an array all have the same type, and an array has arbitrary nonnegative dimension.

**Unions** The union type \( \text{union}(t_1, \ldots, t_n) \) represents values in a polymorphic union type, which is defined as \( t = \alpha_1 \times \ldots \times \alpha_n \). A value of type \( \text{union}(t_1, \ldots, t_n) \) has type \( \tau_j \alpha_1 \times \ldots \times \alpha_n \).

**Unsafe data areas** The unsafe types \( \text{data} \) and \( \text{frame}(t_1, \ldots, t_n) \) represent arbitrary data. Values of type \( \text{data} \) are normally used to represent data aggregates for imperative programming languages, like C, that allow the assignment of values to the data area without regard for the data type. Data areas with the \( \text{data} \) type have no explicit substructure, while values with the \( \text{frame}(t_1, \ldots, t_n) \) type are defined as unsafe records, where the fields in the record are not strongly typed.

**Functions** The function type \( (t_1, \ldots, t_n) \rightarrow t \) includes the functions that return a value of type \( t \), given arguments of types \( t_1, \ldots, t_n \).

**Polymorphic type variables** The variables \( \alpha, \beta, \ldots \) refer to type variables and parameters, used in the polymorphic types below.

**Type application** The type \( \langle t_1, \ldots, t_n \rangle \) applies arguments to a type function, defined as part of a program context (see Chapter 3). If the definition is a parameterized type \( \lambda \alpha_1 \times \ldots \times \alpha_n \rightarrow t \), the type \( \langle t_1, \ldots, t_n \rangle \) is defined as the type \( t_1 \alpha_1 \times \ldots \times \alpha_n \). For example, in a context containing the definition \( \gamma = \lambda \alpha, \beta \rightarrow (\alpha, \beta) \), the type \( \gamma [\mathbb{Z}_3, \mathbb{Z}_3] \) is the same as the type \( \langle \mathbb{Z}_3, \mathbb{Z}_3 \rangle \).

**Universal types** The universal type \( \forall \alpha \times \alpha \rightarrow t \) defines a polymorphic type, where \( t \) must be a function type (see Section 4.6).

**Existential types** The existential type \( \exists \alpha \times \alpha \rightarrow t \) defines a type abstraction. The values in an existential type have the form \( \text{ty_pack}[\exists \alpha_1, \ldots, \alpha_n \rightarrow \alpha, \beta] \). The values in an existential type have the form \( \text{ty_pack}[\exists \alpha_1, \ldots, \alpha_n \rightarrow \alpha, \beta] \).

**Projections** The type projection \( v.i \) is used for values having existential type \( \exists \alpha_1, \ldots, \alpha_n \rightarrow t \). If a value \( v = \text{ty_pack}[\exists \alpha_1, \ldots, \alpha_n \rightarrow \alpha, \beta] \) has type \( \exists \alpha_1, \ldots, \alpha_n \rightarrow t \), then \( v.i \) is equivalent to \( t.i \).

### 2.2.2 Type definitions

Type definitions define parameterized types, unions and frame records. A type definition \( t y d e f \) has the form \( \lambda \alpha_1, \ldots, \alpha_n \rightarrow t y d e f \), where \( t y d e f \) is a simple type definition abstracted over type parameters \( \alpha_1, \ldots, \alpha_n \).

A simple type definition can be a type, a disjoint union, or a frame record type. Disjoint unions are defined on type vectors \( t_1 \oplus \ldots \oplus t_n \). Each type vector \( t_i \) is effectively a tuple space \( (t_1, \ldots, t_n) \). The translation of ML datatypes is straightforward. For example, an ML datatype for labeled binary trees might be declared as follows:

\[
\begin{align*}
\text{type } & \text{'a tree} = \\
& \quad \text{Leaf} \\
& \quad | \text{Node_of } \text{'a tree } * \text{'a } * \text{'a tree}
\end{align*}
\]

\( ^4 \)By \( \tau[\alpha] \) we mean non-capturing substitution of \( t \) for \( \alpha \) in a term \( \tau \). See Section 3.5.3.
CHAPTER 2. THE FIR SYNTAX

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>FIR Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t ::= \mathbb{Z}_{\text{int}}$</td>
<td>Boxed integers</td>
<td>$\text{TyInt}$</td>
</tr>
<tr>
<td>$\text{enum}_i$</td>
<td>Integer enumerations</td>
<td>$\text{TyEnum } i$</td>
</tr>
<tr>
<td>$\mathbb{Z}_{\text{sign}}$</td>
<td>Native integers</td>
<td>$\text{TyRawInt} (\text{pre, sign})$</td>
</tr>
<tr>
<td>$\mathbb{R}_{\text{fpred}}$</td>
<td>Floating-point values</td>
<td>$\text{TyFloat } fpred$</td>
</tr>
<tr>
<td>$\langle t_1, \ldots, t_m \rangle_{\text{tupleclass}}$</td>
<td>Tuple type</td>
<td>$\text{TyTuple} (\text{tupleclass}, [t_1; \cdots; t_m])$</td>
</tr>
<tr>
<td>$t \text{ array}$</td>
<td>Array type</td>
<td>$\text{TyArray } t$</td>
</tr>
<tr>
<td>$\text{data}(tv[t_1, \ldots, t_m], set_1)$</td>
<td>Constructor type</td>
<td>$\text{TyUnion} (tv, [t_1; \cdots; t_m], set_1)$</td>
</tr>
<tr>
<td>$\text{data}$</td>
<td>Unsafe data</td>
<td>$\text{TyRawData}$</td>
</tr>
<tr>
<td>$\text{frame}(tv[t_1, \ldots, t_m])$</td>
<td>Frame record</td>
<td>$\text{TyFrame} (tv, [t_1; \cdots; t_m])$</td>
</tr>
<tr>
<td>$(t_1, \ldots, t_m) \to t$</td>
<td>Function type</td>
<td>$\text{TyFun} ([t_1; \cdots; t_m], t)$</td>
</tr>
<tr>
<td>$\alpha, \beta, \ldots$</td>
<td>Polymorphic type vars</td>
<td>$\text{TyVar } \alpha, \text{TyVar } \beta, \ldots$</td>
</tr>
<tr>
<td>$tv[t_1, \ldots, t_m]$</td>
<td>Type application</td>
<td>$\text{TyApply} (tv, [t_1; \cdots; t_m])$</td>
</tr>
<tr>
<td>$\forall \alpha_1, \ldots, \alpha_m. t$</td>
<td>Universal types</td>
<td>$\text{TyAll} ([\alpha_1; \cdots; \alpha_m], t)$</td>
</tr>
<tr>
<td>$\exists \alpha_1, \ldots, \alpha_m. t$</td>
<td>Existential types</td>
<td>$\text{TyExists} ([\alpha_1; \cdots; \alpha_m], t)$</td>
</tr>
<tr>
<td>$v.i$</td>
<td>Abstract type</td>
<td>$\text{TyProj} (v, i)$</td>
</tr>
</tbody>
</table>

Figure 2.2: The FIR type system

The FIR representation eliminates the constructor names, and forms a disjoint union of tuples.

$$\text{tree} = \Lambda \alpha . () + \langle \text{tree} | \alpha, \text{tree} | \alpha \rangle$$

Frame records are used for representing procedure frames for a language like C. A frame is a doubly-nested record, where the primary field corresponds to each procedure variable in the frame, and the sub-record describes the variable’s parts (for example, pointer variables have two parts: a base pointer that points to the base of the data area, and an integer offset into the data area).

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t ::= \langle t_1, \ldots, t_n \rangle$</td>
<td>List of types</td>
</tr>
<tr>
<td>$R ::= { t_1^i : t_1, \ldots, t_m^i : t_m }$</td>
<td>Record type</td>
</tr>
<tr>
<td>$\text{tydef}_s ::= t$</td>
<td>Type abstraction</td>
</tr>
<tr>
<td>$\bar{t}_1 + \cdots + \bar{t}_m$</td>
<td>Union type</td>
</tr>
<tr>
<td>${ t_1^i : R_1, \ldots, t_m^i : R_m }$</td>
<td>Frame type</td>
</tr>
<tr>
<td>$\text{tydef} ::= \Lambda \alpha_1, \ldots, \alpha_m. \text{tydef}_s$</td>
<td>Polymorphic type</td>
</tr>
</tbody>
</table>

Figure 2.3: FIR type definitions

2.3 FIR expressions

Expressions in the FIR are divided into two classes: the atoms $a$, and general expressions $e$. 
2.3.1 Atoms

The atoms $a$ represent values, including numbers, variables, and basic arithmetic. They are listed in Figure 2.4. Atoms are functional: apart from arithmetic exceptions\(^5\), the order of atom evaluation does not matter. The atoms include the following.

**Numbers**

- The integers int$(i)$ have type $\mathbb{Z}_{31}$.
- The enumeration values $\text{enum}_{i_{\text{prev}}}(i_{\text{value}})$ represent constant values in type $\text{enum}_{i_{\text{bound}}}$. For the atom to be well-formed, $0 \leq i_{\text{value}} < i_{\text{bound}}$ must hold.
- The raw integers $\text{rawint}_{\text{sign}}^{\text{pre}}(r)$ are integer constants with any of the various bit precisions $\text{pre}$, and signedness $\text{sign}$.
- The floating-point numbers $\text{float}_{\text{pre}}(x)$ are constants with any of the various floating-point bit precisions $\text{pre}$.

**Constant constructors** Constant atoms are an optimization defined for allocating zero-arity elements of unions. The atom $\text{const}_{\text{union}}(tv[t_1, \ldots, t_m], \{i\})(tv, i)$ has union type $\text{union}(tv[t_1, \ldots, t_m], \{i\})$ if the $i^{th}$ case of the union defined by $tv$ has zero arity.

**Frame record values** The labels $\text{label}(tv, l^l, l^s, r)$ represent offsets into a frame record. The type variable $tv$ is the name of the frame type definition, the label $l^l$ is the field name, the label $l^s$ is the sub-field name, and $r$ is an additional offset into the field.

Similarly, the atom $\text{sizeof}(tv_1, \ldots, tv_n, r)$ represents the combined size of a list of frames $tv_1, \ldots, tv_n$ plus a constant $r$.

**Variables** The variables $v$ represent values defined in the program environment, described in Chapter 3. Variables are immutable: the FIR does not include a variable assignment operation.

**Polymorphic operations**

- The $\text{ty\_apply}[l](v, t_1, \ldots, t_m)$ atom is a type application of a polymorphic value $v$ to type arguments $t_1, \ldots, t_m$. For the application to be well-formed, the variable $v$ must have universal type $\forall \alpha_1, \ldots, \alpha_m. u$; the atom has type $t = u[t_1/\alpha_1, \ldots, t_m/\alpha_m]$.
- The $\text{ty\_pack}[l](v, t_1, \ldots, t_m)$ atom performs type abstraction. It has type $t = \exists \alpha_1, \ldots, \alpha_m. u$ when $v$ has type $u[t_1/\alpha_1, \ldots, t_m/\alpha_m]$.
- The $\text{ty\_unpack}(v)$ atom is the elimination form for type abstraction packages. If $v$ has existential type $\exists \alpha_0, \ldots, \alpha_{m-1}. u$, the atom has type $u[v.0/\alpha_0, \ldots, v.(m-1)/\alpha_{m-1}]$. The types $v.i$ represent the type parameter $t_i$ in the original pack operation.

**Arithmetic** There are two forms for arithmetic: unary operations $\text{unop} a$, and binary operations $a_1 \text{\ binop} a_2$. The operator tables are listed in Figures 2.5–2.7.

---

\(^5\)A notable arithmetic exception is division by zero.
FIR Safety

CHAPTER 2. THE FIR SYNTAX

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
<th>FIR Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a ) :=</td>
<td>( \text{enum}<em>{t</em>{\text{value}}}() )</td>
<td>Enumerated value</td>
</tr>
<tr>
<td>( \text{int}(i) )</td>
<td>Boxed integer constants</td>
<td>( \text{AtomInt} i )</td>
</tr>
<tr>
<td>( \text{rawint}^\text{sign}(r) )</td>
<td>Raw integer constants</td>
<td>( \text{AtomRawInt} r )</td>
</tr>
<tr>
<td>( \text{float}_{\text{fp}}(x) )</td>
<td>Floating-point constants</td>
<td>( \text{AtomFloat} x )</td>
</tr>
<tr>
<td>( \text{label}(tv, l^f, l^f, r) )</td>
<td>Offset into a frame record</td>
<td>( \text{AtomLabel}(tv, l^f, l^f, r) )</td>
</tr>
<tr>
<td>( \text{sizeof}(tv_1, \ldots, tv_n, r) )</td>
<td>Size of frames, plus ( r )</td>
<td>( \text{AtomSizeof}([tv_1; \ldots; tv_n], r) )</td>
</tr>
<tr>
<td>( \text{const}[t](tv, i) )</td>
<td>Constant union constructor</td>
<td>( \text{AtomConst}(t, tv, i) )</td>
</tr>
<tr>
<td>( v )</td>
<td>Variables</td>
<td>( \text{AtomVar} v )</td>
</tr>
<tr>
<td>( \text{ty_apply}(t)(a, t_1, \ldots, t_m) )</td>
<td>Type application</td>
<td>( \text{AtomTyApply}(a, t, [t_1; \ldots; t_m]) )</td>
</tr>
<tr>
<td>( \text{ty_pack}(t)(v, t_1, \ldots, t_m) )</td>
<td>Existential packing</td>
<td>( \text{AtomTyPack}(v, t, [t_1; \ldots; t_m]) )</td>
</tr>
<tr>
<td>( \text{ty_unpack}(v) )</td>
<td>Existential unpacking</td>
<td>( \text{AtomTyUnpack} v )</td>
</tr>
<tr>
<td>( \text{unop} a )</td>
<td>Unary operation</td>
<td>( \text{AtomUnop}(\text{unop}, a) )</td>
</tr>
<tr>
<td>( a_1 \text{ binop } a_2 )</td>
<td>Binary operation</td>
<td>( \text{AtomBinop}(\text{binop}, a_1, a_2) )</td>
</tr>
</tbody>
</table>

Figure 2.4: FIR atoms

**Unary operators**

The unary operators are shown in Figure 2.5. We use the following meta-notation in the table. For an operator \( \text{unop} \), the term \( \text{arg} (\text{unop}) \) specifies the type of the argument of the operator (in the \( \text{Arg} \) column), and the term \( \text{res} (\text{unop}) \) specifies the type of the result (in the \( \text{Result} \) column).

Most of the operators are for performing arithmetic with various bit precisions. The exception to this is \( t_d \leftarrow t_s \), which performs a runtime coercion. While we do not list the coercions explicitly here, the coercions are mainly between numerical arguments. Some coercions require runtime checks to ensure safety.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
<th>Arg</th>
<th>Description</th>
<th>FIR Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sim_{\text{z}_{31}} )</td>
<td>( \mathbb{Z}_{31} )</td>
<td>( \mathbb{Z}_{31} )</td>
<td>Unary negation</td>
<td>( \text{UMinusIntOp} )</td>
</tr>
<tr>
<td>( \sim_{\text{z}_{\text{pre}}} )</td>
<td>( \mathbb{Z}_{\text{pre}} )</td>
<td>( \mathbb{Z}_{\text{pre}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sim_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{lnot}<em>{\mathbb{Z}</em>{31}} )</td>
<td>( \mathbb{Z}_{31} )</td>
<td>( \mathbb{Z}_{31} )</td>
<td>Bitwise negation</td>
<td>( \text{NotIntOp} )</td>
</tr>
<tr>
<td>( \text{lnot}<em>{\mathbb{R}</em>{\text{pre}}} )</td>
<td>( \mathbb{R}_{\text{pre}} )</td>
<td>( \mathbb{R}_{\text{pre}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{bitfield}_{\text{pre}, \text{sign}} )</td>
<td>( \mathbb{Z}_{\text{sign}} )</td>
<td>( \mathbb{Z}_{\text{sign}} )</td>
<td>Extract bitfield at offset ( i ) (size ( j ))</td>
<td>( \text{RawBitFieldOp}(\text{pre}, \text{sign}, i, j) )</td>
</tr>
<tr>
<td>( \text{abs}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>Absolute value</td>
<td>( \text{AbsFloatOp}_{\text{fp}} )</td>
</tr>
<tr>
<td>( \text{sin}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>Sine</td>
<td>( \text{SinOp}_{\text{fp}} )</td>
</tr>
<tr>
<td>( \text{cos}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>Cosine</td>
<td>( \text{CosOp}_{\text{fp}} )</td>
</tr>
<tr>
<td>( \text{sqrt}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>( \mathbb{R}_{\text{fp}} )</td>
<td>Square root</td>
<td>( \text{SqrtOp}_{\text{fp}} )</td>
</tr>
<tr>
<td>( (t_d \leftarrow t_s) )</td>
<td>( t_d )</td>
<td>( t_s )</td>
<td>Coerce between types (safely)</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

Figure 2.5: FIR unary operators
Binary operators

There are two classes of binary operators: comparisons, and arithmetic, shown in Figures 2.6 and 2.7, respectively. We use the following meta-notation in the tables. For an operator $\text{binop}$, the term $\text{arg}_i(\text{binop})$ specifies the type of argument $i$ of the operator. The index $i$ is either 1 or 2, and the type is listed in the $\text{Arg}_i$ column. The term $\text{res}(\text{binop})$ specifies the type of the result, listed in the $\text{Result}$ column.

The comparison operators are shown in Figure 2.6. In most cases, the result of a comparison has the type $\textbf{enum}_2$, which is used to represent Boolean values. The $\textbf{cmp}_p$ operators are an exception. The result of a comparison $i \textbf{cmp}_p j$ is negative if $i < j$ (using the specified precision and signedness), positive if $i > j$, and zero if $i = j$.

There are also two non-arithmetic comparisons. The operator $=^i$ compares two values of type $t$. The type $t$ is arbitrary, and we do not specify the semantics precisely. For any comparison $a_1 =^i a_2$, the result is not true if $a_1 \neq a_2$. The comparison $a_1 \neq^i a_2$ is the same as $\neg(a_1 =^i a_2)$.

The operators for binary arithmetic and bitwise operations are shown in Figure 2.7.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Description</th>
<th>FIR Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$=_{\text{z}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{z}_{31}$</td>
<td>Test for equality</td>
<td>$\text{Eq} \text{Int Op}$</td>
</tr>
<tr>
<td>$=<em>{\text{z}</em>{\text{ZX}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>Test for equality</td>
<td>$\text{Eq} \text{RawInt Op} (\text{pre}, \text{sign})$</td>
</tr>
<tr>
<td>$=<em>{\text{R}</em>{\text{pre}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{R}_{\text{pre}}$</td>
<td>$\text{R}_{\text{pre}}$</td>
<td>Test for equality</td>
<td>$\text{Eq} \text{Float Op} \text{ fpre}$</td>
</tr>
<tr>
<td>$\not=_{\text{z}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{z}_{31}$</td>
<td>Test for inequality</td>
<td>$\text{Nq} \text{Int Op}$</td>
</tr>
<tr>
<td>$\not=<em>{\text{z}</em>{\text{ZX}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>Test for inequality</td>
<td>$\text{Nq} \text{RawInt Op} (\text{pre}, \text{sign})$</td>
</tr>
<tr>
<td>$&lt;_{\text{z}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{z}_{31}$</td>
<td>Test for less than</td>
<td>$\text{Lt} \text{Int Op}$</td>
</tr>
<tr>
<td>$&lt;<em>{\text{z}</em>{\text{ZX}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>Test for less than</td>
<td>$\text{Lt} \text{RawInt Op} (\text{pre}, \text{sign})$</td>
</tr>
<tr>
<td>$\leq_{\text{z}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>Test for less than or equal</td>
<td>$\text{Le} \text{Int Op}$</td>
</tr>
<tr>
<td>$\leq_{\text{z}_{\text{ZX}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>Test for less than or equal</td>
<td>$\text{Le} \text{RawInt Op} (\text{pre}, \text{sign})$</td>
</tr>
<tr>
<td>$&gt;_{\text{z}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{z}_{31}$</td>
<td>Test for greater than</td>
<td>$\text{Gt} \text{Int Op}$</td>
</tr>
<tr>
<td>$&gt;<em>{\text{z}</em>{\text{ZX}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>Test for greater than</td>
<td>$\text{Gt} \text{RawInt Op} (\text{pre}, \text{sign})$</td>
</tr>
<tr>
<td>$\geq_{\text{z}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{R}_{\text{pre}}$</td>
<td>$\text{R}_{\text{pre}}$</td>
<td>Test for greater than or equal</td>
<td>$\text{Gt} \text{Int Op}$</td>
</tr>
<tr>
<td>$\geq_{\text{z}_{\text{ZX}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{z}_{31}$</td>
<td>Test for greater than or equal</td>
<td>$\text{Gt} \text{RawInt Op} (\text{pre}, \text{sign})$</td>
</tr>
<tr>
<td>$\textbf{cmp}_{\text{z}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{ML}-$style comparison</td>
<td>$\text{CmpInt Op}$</td>
</tr>
<tr>
<td>$\textbf{cmp}<em>{\text{z}</em>{\text{ZX}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{z}_{\text{sign}}$</td>
<td>$\text{ML}-$style comparison</td>
<td>$\text{CmpRawInt Op} (\text{pre}, \text{sign})$</td>
</tr>
<tr>
<td>$\textbf{cmp}<em>{\text{R}</em>{\text{pre}}}$</td>
<td>$\textbf{enum}_2$</td>
<td>$\text{z}_{31}$</td>
<td>$\text{R}_{\text{pre}}$</td>
<td>$\text{ML}-$style comparison</td>
<td>$\text{CmpFloat Op} \text{ fpre}$</td>
</tr>
<tr>
<td>$=_{^i}$</td>
<td>$\textbf{enum}_2$</td>
<td>$t$</td>
<td>$t$</td>
<td>Test for pointer equality</td>
<td>$\text{Eq} \text{EqOp} \text{ i}$</td>
</tr>
<tr>
<td>$\neq_{^i}$</td>
<td>$\textbf{enum}_2$</td>
<td>$t$</td>
<td>$t$</td>
<td>Test for pointer inequality</td>
<td>$\text{Nq} \text{EqOp} \text{ i}$</td>
</tr>
</tbody>
</table>

Figure 2.6: FIR binary comparison operators

2.3.2 Expressions

The expressions $e$ are shown in Figure 2.8. Both scope and control-flow are explicit.

Figure 2.7: FIR binary operators
<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Description</th>
<th>FIR Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Addition</td>
<td>PlusIntOp</td>
</tr>
<tr>
<td>+</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>PlusIntOp</td>
</tr>
<tr>
<td></td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Subtraction</td>
<td>MinusIntOp</td>
</tr>
<tr>
<td></td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>MinusIntOp</td>
</tr>
<tr>
<td>*</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Multiplication</td>
<td>MulIntOp</td>
</tr>
<tr>
<td>/</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Division</td>
<td>DivIntOp</td>
</tr>
<tr>
<td>rem</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Remainder</td>
<td>RemIntOp</td>
</tr>
<tr>
<td>lsl</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Logical shift left</td>
<td>LslIntOp</td>
</tr>
<tr>
<td>lsr</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Logical shift right</td>
<td>LsrIntOp</td>
</tr>
<tr>
<td>asr</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Arithmetic shift right</td>
<td>AsrIntOp</td>
</tr>
<tr>
<td>sl</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Shift left</td>
<td>SlRawIntOp</td>
</tr>
<tr>
<td>sr</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Shift right</td>
<td>SrRawIntOp</td>
</tr>
<tr>
<td>and</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Logical and</td>
<td>AndIntOp</td>
</tr>
<tr>
<td>and.$\ll$</td>
<td>enum$_i$.</td>
<td>enum$_i$.</td>
<td>enum$_i$.</td>
<td>Logical or</td>
<td>OrIntOp</td>
</tr>
<tr>
<td>or</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Logical xor</td>
<td>XorIntOp</td>
</tr>
<tr>
<td>xnor</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Logical xor</td>
<td>XorRawIntOp</td>
</tr>
<tr>
<td>xor.$\ll$</td>
<td>enum$_i$.</td>
<td>enum$_i$.</td>
<td>enum$_i$.</td>
<td>Logical xor</td>
<td>XorIntOp</td>
</tr>
<tr>
<td>xor.$\ll$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Logical xor</td>
<td>XorRawIntOp</td>
</tr>
<tr>
<td>max</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Maximum of two values</td>
<td>MaxIntOp</td>
</tr>
<tr>
<td>max.$\ll$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Maximum of two values</td>
<td>MaxIntOp</td>
</tr>
<tr>
<td>max.$\ll$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Maximum of two values</td>
<td>MaxIntOp</td>
</tr>
<tr>
<td>min</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Minimum of two values</td>
<td>MinIntOp</td>
</tr>
<tr>
<td>min.$\ll$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Minimum of two values</td>
<td>MinIntOp</td>
</tr>
<tr>
<td>min.$\ll$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Minimum of two values</td>
<td>MinIntOp</td>
</tr>
<tr>
<td>atan2</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Arc tangent</td>
<td>Atan2Op</td>
</tr>
<tr>
<td>set.bitfield$_{pre,\text{sign}}^{i,j}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>$Z_{31}$</td>
<td>Set bit field at offset</td>
<td>RawSetBitFieldOp</td>
</tr>
</tbody>
</table>

Figure 2.7: FIR binary operators

**LetAtom** The let $v \colon t = a$ in $e$ expression forms a new scope, where the variable $v$ is bound to the value of the atom expression $a$ in the expression $e$. For the expression to be well-formed, the atom must have type $t$, and the expression $e$ must be well-formed for an arbitrary value $v$ of type $t$.

**LetExt** The external-call expression let $v \colon (t_1) (a_1, \ldots, a_n)$ in $e$ is used to provide access to the runtime and operating system. The string “s” represents the name of a runtime function to be called.
with arguments \( a_1, \ldots, a_n \). For the expression to be well-formed, the runtime function "s" must have type \( t_s = (u_1, \ldots, u_n) \to t_v \), each atom \( a_i \) must have type \( u_i \), and \( e \) must be well-formed given a value \( v \) with type \( t_v \).

**TailCall** The tail-call \( a(a_1, \ldots, a_n) \) represents a function call to the function \( a \), with arguments \( a_1, \ldots, a_n \). For the tail-call to be well-formed, the function \( a \) must have some type \( (u_1, \ldots, u_n) \to \text{enum}_0 \), and each argument \( a_i \) must have type \( u_i \).

The return type of the function is the empty type \( \text{enum}_0 \). There is no syntactic mechanism for using the return value of a function, and functions never return.

**SpecialCall** The special-call special tailop represents a call for process migration, or one of the atomic transaction operations. The tailop operations are shown in Figure 2.9. The special-call operations are described in detail in Chapter 8. The typing rules are defined in Section 4.8.2, and the operational semantics are defined in Section 5.4.

- The operator \( \text{migrate} [i, a_p, a_o] \) \( a_{\text{fun}}(a_1, \ldots, a_n) \) defines a process migration. The argument \( (a_p, a_o) \) specifies the destination (for example, the name of a migration server), and the argument \( a_{\text{fun}}(a_1, \ldots, a_n) \) specifies a function to be called once the process migrates, along with its arguments.

- The operator \( \text{atomic} a_{\text{fun}}(a_{\text{const}}, a_1, \ldots, a_n) \) specifies entry into an atomic transaction. This operation effectively takes a process checkpoint, which can be restored if the transaction is aborted. Once the transaction is entered, the function \( a_{\text{fun}} \) is called with arguments \( a_{\text{const}}, a_1, \ldots, a_n \).

- The operator \( \text{rollback} [a_{\text{level}}, a_{\text{const}}] \) is used to abort a transaction. The atom \( a_{\text{level}} \) is the name of the transaction to be aborted (the transaction checkpoints are saved in a list). Transactions are restarted on failure, and the atom \( a_{\text{const}} \) is a new argument to be passed to the transaction entry point when the transaction is restarted. For example, if the transaction was entered with the operator \( \text{atomic} a_{\text{fun}}(a_{\text{const}}, a_1, \ldots, a_n) \), and the transaction is aborted with the \( \text{rollback} [a'_{\text{level}}, a'_{\text{const}}] \), the transaction is resumed with the tail-call \( a_{\text{fun}}(a'_{\text{const}}, a_1, \ldots, a_n) \).

- The operator \( \text{commit} [a_{\text{level}}] \) \( a_{\text{fun}}(a_1, \ldots, a_n) \) commits the transaction identified by \( a_{\text{level}} \). The transaction checkpoint identified by \( a_{\text{level}} \) is discarded, and the function \( a_{\text{fun}} \) is called with arguments \( a_1, \ldots, a_n \).

**Match** The match statement \( \text{match } a \text{ with } [s_i \mapsto e_i]_i^n \) is a pattern match of an integer against multiple sets. Each match case \( s_i \mapsto e_i \) specifies an integer (or raw integer) set \( s_i \) and an expression \( e_i \) to be evaluated if \( a \in s_i \). Evaluation is ordered and total. Evaluation chooses the first match that succeeds, and the match statement is well-formed only if there is a match case for any possible value of \( a \).

**Data aggregates** The aggregate data areas include tuples, arrays, elements in a union type, raw data, and frames. The \( \text{let } v = \text{alloc } e \) expression allocates a data aggregate, using one of the \( \text{alloc} \) forms shown in Figure 2.9.

Values are projected from an aggregate data area using the \( \text{let } v : t = a_1[a_2] \text{ in } e \) expression. For the expression to be well-formed, \( a_1 \) must be an aggregate, and \( a_2 \) must be a valid index into the aggregate. All fields in aggregates are mutable. The \( a_1[a_2] : t \leftarrow a_3 \) \( e \) expression assigns value \( a_3 \) to field \( a_2 \) in aggregate \( a_1 \).
Global values The global values are a set of global mutable values (normally used to represent global variables in languages like C). Global values are not directly accessible as atoms. The value in global location l can be retrieved with the expression let v: t = global l in e, and assigned with the expression global l: t = a; e.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
<th>FIR Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>e ::= let v: t = a in e</td>
<td>Primitive operations</td>
<td>LetAtom (v, t, a, e)</td>
</tr>
<tr>
<td></td>
<td>let v: t_v = (&quot;s&quot;: t_s)(a_1, ..., a_n) in e</td>
<td>Calls to the runtime</td>
</tr>
<tr>
<td></td>
<td>a(a_1, ..., a_n)</td>
<td>Tail-call</td>
</tr>
<tr>
<td></td>
<td>special tailop</td>
<td>Special tail-call</td>
</tr>
<tr>
<td></td>
<td>match a with [s_i → e_i]</td>
<td>Case analysis</td>
</tr>
<tr>
<td></td>
<td>let v = alloc in e</td>
<td>Allocation</td>
</tr>
<tr>
<td></td>
<td>let v: t = a_1[a_2] in e</td>
<td>Load from heap</td>
</tr>
<tr>
<td></td>
<td>a_1[a_2]: t ← a_3; e</td>
<td>Store into heap</td>
</tr>
<tr>
<td></td>
<td>let v: t = global l in e</td>
<td>Load from global</td>
</tr>
<tr>
<td></td>
<td>global l: t ← a; e</td>
<td>Store into global</td>
</tr>
</tbody>
</table>

Figure 2.8: FIR expressions

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
<th>FIR Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>unaryop ::= −</td>
<td></td>
<td>Unary operations</td>
</tr>
<tr>
<td>binop ::= +</td>
<td></td>
<td>Binary operations</td>
</tr>
<tr>
<td>alloc ::= ⟨a_1, ..., a_n⟩ tupleωclass : t</td>
<td>Tuple allocation</td>
<td>AllocTuple ⟨...⟩</td>
</tr>
<tr>
<td></td>
<td>union⟨tv[a_1, ..., a_n], i⟩ : t</td>
<td>Union allocation</td>
</tr>
<tr>
<td></td>
<td>array⟨a_size, a_init⟩ : t</td>
<td>Array allocation</td>
</tr>
<tr>
<td></td>
<td>malloc(a) : t</td>
<td>Rawdata allocation</td>
</tr>
<tr>
<td></td>
<td>frame⟨tv[t_1, ..., t_m]⟩</td>
<td>Allocate a frame</td>
</tr>
<tr>
<td>tailop ::= migrate [i, a_p, a_o] α_fun(a_1, ..., a_n)</td>
<td>System migration</td>
<td>TailSysMigrate ⟨...⟩</td>
</tr>
<tr>
<td></td>
<td>atomic α_fun(a_const, a_1, ..., a_n)</td>
<td>Transaction entry</td>
</tr>
<tr>
<td></td>
<td>rollback [α_level, α_const]</td>
<td>Transaction rollback</td>
</tr>
<tr>
<td></td>
<td>commit [α_level] α_fun(a_1, ..., a_n)</td>
<td>Transaction commit</td>
</tr>
</tbody>
</table>

Figure 2.9: FIR operators
Chapter 3

Judgments

All judgments, including type and well-formedness judgments, are defined with respect to an environment \( \Gamma \), which we also call a context. The environment contains both variable declarations of the form \( v: t \), and variable definitions of the form \( v: t = b \). The declaration \( v: t \) specifies that a variable \( v \) has an unspecified value of type \( t \). The definition \( v: t = b \) specifies that variable \( v \) has the value \( b \), and \( b \) has type \( t \).

3.1 Heap and store values

The definitions in the context use values of two sorts: heap values \( h \), and store values \( b \). The heap values represent atoms that have been fully evaluated. In general, an atom evaluates to a number, a label, or a variable that represents a store value (described below).

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h ) := ( \text{enum}<em>{\text{int}}(t</em>{\text{value}}) )</td>
<td>Enumerated values</td>
</tr>
<tr>
<td>( \text{int}(i) )</td>
<td>Boxed integer constants</td>
</tr>
<tr>
<td>( \text{rawint}_{\text{sign}}(r) )</td>
<td>Raw integer constants</td>
</tr>
<tr>
<td>( \text{float}_{\text{pre}}(x) )</td>
<td>Floating-point constants</td>
</tr>
<tr>
<td>( \text{label}(tv, I^l, I^r, r) )</td>
<td>Offset into a frame record</td>
</tr>
<tr>
<td>( v )</td>
<td>Variables</td>
</tr>
</tbody>
</table>

Figure 3.1: Heap values

The store values are the values in a program store. These include heap values, functions, “packed” values with existential type, and data in each of the aggregate data types: tuples, arrays, elements of a union type, rawdata, and frames.

Functions are universally quantified, with type parameters \( \alpha_1, \ldots, \alpha_m \), and actual parameters \( v_1, \ldots, v_n \). Elements of a union type are like tagged tuples: they include the type variable \( tv \) that defines the union space, an index \( i \) for the union tag, and the elements of the tuple being tagged. Elements of type data are represented abstractly using the form \( (c) \); the elements in the data area are not explicitly described. Frame

\(^1\)We allow \( m = 0 \) here; that is, a function may or may not have any type parameters.
data is also represented abstractly with the form \( \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle \), where \( \text{frame}(tv[t_1, \ldots, t_n]) \) is the type of the frame.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b ::= h )</td>
<td>Heap values</td>
</tr>
<tr>
<td>(</td>
<td>\Lambda \alpha_1, \ldots, \alpha_m. \lambda v_1, \ldots, v_n. e )</td>
</tr>
<tr>
<td>(</td>
<td>\text{ty.pack}[t](v, t_1, \ldots, t_m) )</td>
</tr>
<tr>
<td>(</td>
<td>\langle h_1, \ldots, h_n \rangle )</td>
</tr>
<tr>
<td>(</td>
<td>tv_i(h_1, \ldots, h_n) )</td>
</tr>
<tr>
<td>(</td>
<td>\langle c \rangle )</td>
</tr>
<tr>
<td>(</td>
<td>\langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle )</td>
</tr>
</tbody>
</table>

Figure 3.2: Store values

### 3.2 Kinds

The program types are also defined/declared as part of the context \( \Gamma \). For presentation purposes, we classify the program types with kinds. The program kinds are shown in Figure 3.3. The kind \( k_s \) classifies the type definitions \( \text{tydef} \) as “small” types \( \omega \), “large” types \( \Omega \), disjoint unions \( \omega_{\text{union}[n]} \), and frames \( \omega_{\text{frame}} \).

The distinction between small and large types is necessary to assist the garbage collector. The raw integers and floating-point values are large, and all other types are small. Values of small type fit into a single machine word, and small values are tagged so the garbage collector can distinguish between pointers and integers. In general, polymorphic functions and data structures can be instantiated with small types, but not large types.

The general kind \( k = \omega^m \rightarrow k_s \) represents a parameterized type definition \( \text{tydef} \). The number of parameters \( m \) may be any nonnegative integer. If \( m = 0 \), we often omit the type parameters.

\[
\begin{align*}
k_s & ::= \omega \\
| & \Omega \\
| & \omega_{\text{union}[n]} \\
| & \omega_{\text{frame}} \\

k & ::= \omega^m \rightarrow k_s
\end{align*}
\]

Figure 3.3: Kinds

### 3.3 Contexts

A program context \( \Gamma \) is defined as a set of mutually-recursive declarations and definitions, as shown in Figure 3.4. There are three forms of definitions.

1. The type definition \( tv : k = \text{tydef} \) defines a type named \( tv \), having kind \( k \), and value \( \text{tydef} \).
2. The variable definition \( v : t = b \) defines a variable named \( v \), with type \( t \) and store value \( b \).
3. The global definition $l: t = h$ defines a global label $l$, with type $t$ and heap value $h$.

For each definition form there is a corresponding declaration form, with syntax as described in Figure 3.4.

We assume that each variable, type variable, and global label in a context is defined/declared at most once, and we use alpha-renaming throughout this paper to rename variables as appropriate.

$$
def ::= v: t \quad \text{Variable declaration}
| v: t = b \quad \text{Variable definition}
| l: t \quad \text{Global declaration}
| l: t = h \quad \text{Global definition}
| tv: k \quad \text{Type declaration}
| tv: k = typedef \quad \text{Type definition}
$$

$$\Gamma ::= \epsilon \quad \text{Empty environment}
| \Gamma, def \quad \text{Adding a definition}$$

Figure 3.4: Program contexts

Note that we also use the meta-variable $d$ to represent a definition or declaration $def$.

### 3.4 Judgments

The judgments are shown in Figure 3.5. The judgment $\Gamma \vdash \phi$ specifies that the context $\Gamma$ is well-formed. As we show in Section 6.1, a context is well-formed if all of its declarations and definitions are well-formed. For each declaration $v: t$ and definition $v: t = b$, the term $t$ must be a well-formed type, and the value $b$ must have type $t$. Similarly, all type and global definitions in $\Gamma$ must be well-formed.

The judgment $\Gamma \vdash typedef_1 = typedef_2 : k$ is a type definition equality judgment. When the judgment is true, $typedef_1$ and $typedef_2$ have the specified kind, and they are equal. There is no separate membership judgment $\Gamma \vdash typedef : k$. However, we will often use the shorthand form $\Gamma \vdash typedef : k$ to stand for $\Gamma \vdash typedef = typedef : k$.

The four judgments $\Gamma \vdash l : t$, $\Gamma \vdash a : t$, $\Gamma \vdash b : t$, $\Gamma \vdash e : t$ express typing relations for labels, atoms, store values, and expressions, respectively.

There are also several auxiliary judgment forms. The $\Gamma \vdash tailop : \text{special} t$ judgment specifies that $tailop$ is a special-call operator returning type $t$. The $\Gamma \vdash alloc : \text{alloc} t$ judgment specifies that $alloc$ is an allocation operator for a value of type $t$. The record judgment $\Gamma \vdash R : \text{record}$ specifies that $R$ is a valid record type that can be used as a subrecord in a frame.

### 3.5 Free variables and substitution

In order to define the typing rules and operational semantics, we first need to define type substitution.
### CHAPTER 3. JUDGMENTS

#### 3.5.1 Domain of context $\Gamma$

The **domain** $\text{dom}(\Gamma)$ of a context $\Gamma$ is the set of type variables, variables, and global labels that are declared and defined. We use $*$ below as a placeholder for any syntactically valid term.

\[
\begin{align*}
\text{dom}(e) &= \{ \} \\
\text{dom}(\Gamma, v: *) &= \text{dom}(\Gamma) \cup \{ v \} \\
\text{dom}(\Gamma, v: = *) &= \text{dom}(\Gamma) \cup \{ v \} \\
\text{dom}(\Gamma, l: *) &= \text{dom}(\Gamma) \cup \{ l \} \\
\text{dom}(\Gamma, l: = *) &= \text{dom}(\Gamma) \cup \{ l \} \\
\text{dom}(\Gamma, t: *) &= \text{dom}(\Gamma) \cup \{ tv \} \\
\text{dom}(\Gamma, tv: = *) &= \text{dom}(\Gamma) \cup \{ tv \}
\end{align*}
\]

#### 3.5.2 Free variables

The **free variables** $FV(\Gamma)$ of a context are defined inductively. To simplify the presentation, we define the free-variables of a term as a set of all the variables, including type variables, normal variables, and global labels that are free in the term.

The free variables of a type are shown in Figure 3.6.

The free variables of a type definition are shown in Figure 3.7. The free variables of a record do not include the frame labels.

The free variables of an atom are shown in Figure 3.8.

The free variables of unary operators are defined as follows.

\[
FV(\text{unop}) ::= \begin{cases} 
FV(t_d) \cup FV(t_s) & \text{if } \text{unop} = (t_d \leftarrow t_s) \\
\{ \} & \text{otherwise}
\end{cases}
\]

The free variables of binary operators are defined as follows.

\[
FV(\text{binop}) ::= \begin{cases} 
FV(t) & \text{if } \text{binop} = =_t \text{ or } \text{binop} = \neq_t \\
\{ \} & \text{otherwise}
\end{cases}
\]
3.5. FREE VARIABLES AND SUBSTITUTION

The free variables of an expression are shown in Figure 3.9.

The expression free variables use the definition of free variables for special-calls (shown in Figure 3.10) and allocation operations (shown in Figure 3.11).

The free variables of store values are shown in Figure 3.12.

The free variables of context declarations and definitions are shown in Figure 3.13.

The free variables of a context $\Gamma$ are the free variables of all the definitions, excluding the variables being declared.

$$FV(\Gamma) = \left( \bigcup_{d \in \Gamma} FV(d) \right) - \text{dom}(\Gamma)$$
\[ a \quad FV(a) \]

1. `enum_{i\text{value}}(t)`
   \[
   \{ \}
   \]
2. `int(i)`
   \[
   \{ \}
   \]
3. `rawint_{pre}(r)`
   \[
   \{ \}
   \]
4. `float_{pre}(x)`
   \[
   \{ \}
   \]
5. `label(t, l', l', r)`
   \[
   \{ t \}
   \]
6. `sizeof(t_1, \ldots, t_n, r)`
   \[ \{ t_1, \ldots, t_n \} \]
7. `const[t](t, i)`
   \[ \{ t \} \cup FV(t) \]
8. `v`
   \[ \{ v \} \]
9. `ty\_apply[t](a, t_1, \ldots, t_m)`
   \[ FV(t) \cup FV(a) \cup \bigcup_{i=1}^m FV(t_i) \]
10. `ty\_pack[t](v, t_1, \ldots, t_m)`
    \[ FV(t) \cup \{ v \} \cup \bigcup_{i=1}^m FV(t_i) \]
11. `ty\_unpack(v)`
    \[ \{ v \} \]
12. `unop a`
    \[ FV(a) \cup FV(unop) \]
13. `a_1 binop a_2`
    \[ FV(a_1) \cup FV(a_2) \cup FV(binop) \]

**Figure 3.8: Atom free variables**

\[ e \quad FV(e) \]

1. `let v: t = a in e`
   \[ FV(t) \cup FV(a) \cup (FV(e) - \{ v \}) \]
2. `let v: t = ("s": t_s)(a_1, \ldots, a_n) in e`
   \[ FV(t_v) \cup FV(t_s) \cup \bigcup_{i=1}^n FV(a_i) \cup (FV(e) - \{ v \}) \]
3. `a(a_1, \ldots, a_n)`
   \[ FV(a) \cup \bigcup_{i=1}^n FV(a_i) \]
4. `special tailop`
   \[ FV(tailop) \]
5. `match a with s_1 \mapsto e_1, \ldots, s_n \mapsto e_n`
   \[ FV(a) \cup \bigcup_{i=1}^n FV(e_i) \]
6. `let v = alloc in e`
   \[ FV(alloc) \cup (FV(e) - \{ v \}) \]
7. `let v: t = a_1[a_2] in e`
   \[ FV(t) \cup FV(a_1) \cup FV(a_2) \cup (FV(e) - \{ v \}) \]
8. `a_1[a_2]: t \leftarrow a_3; e`
   \[ FV(a_1) \cup FV(a_2) \cup FV(t) \cup FV(a_3) \cup FV(e) \]
9. `let v: t = global/ in e`
   \[ FV(t) \cup \{ l \} \cup (FV(e) - \{ v \}) \]
10. `global/ t \leftarrow a; e`
    \[ \{ l \} \cup FV(t) \cup FV(a) \cup FV(e) \]

**Figure 3.9: Expression free variables**

\[ alloc \quad FV(alloc) \]

1. `<a_1, \ldots, a_n>_{\text{tuple, class}}: t`
   \[ FV(t) \cup \bigcup_{i=1}^n FV(a_i) \]
2. `union(tv[a_1, \ldots, a_m], t): t`
   \[ FV(t) \cup \{ tv \} \cup \bigcup_{i=1}^m FV(a_i) \]
3. `array(a_{size}, a_{init}): t`
   \[ FV(t) \cup FV(a_{size}) \cup FV(a_{init}) \]
4. `malloc(a): t`
   \[ FV(t) \cup FV(a) \]
5. `frame(tv[t_1, \ldots, t_m])`
   \[ \{ tv \} \cup \bigcup_{i=1}^m FV(t_i) \]

**Figure 3.10: Allocation operator free variables**
3.5. FREE VARIABLES AND SUBSTITUTION

<table>
<thead>
<tr>
<th>tailop</th>
<th>FV(tailop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>migrate [i, a_p, a_o] a_{fun}(a_1, \ldots, a_n)</td>
<td>(FV(a_p) \cup FV(a_o) \cup FV(a_{fun}) \cup \bigcup_{i=1}^n FV(a_i))</td>
</tr>
<tr>
<td>atomic a_{fun(a_{const}, a_1, \ldots, a_n)}</td>
<td>(FV(a_{fun}) \cup FV(a_{const}) \cup \bigcup_{i=1}^n FV(a_i))</td>
</tr>
<tr>
<td>rollback [a_{level}, a_{const}]</td>
<td>(FV(a_{level}) \cup FV(a_{const}))</td>
</tr>
<tr>
<td>commit [a_{level}] a_{fun}(a_1, \ldots, a_n)</td>
<td>(FV(a_{level}) \cup FV(a_{fun}) \cup \bigcup_{i=1}^n FV(a_i))</td>
</tr>
</tbody>
</table>

Figure 3.11: Special-call free variables

<table>
<thead>
<tr>
<th>b</th>
<th>FV(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>FV(h)</td>
</tr>
<tr>
<td>(\lambda x_1, \ldots, x_m. \lambda v_1, \ldots, v_n. e)</td>
<td>(FV(e) - {\alpha_1, \ldots, \alpha_m, v_1, \ldots, v_n})</td>
</tr>
<tr>
<td>typack[\tau](v, t_1, \ldots, t_m)</td>
<td>(FV(\tau) \cup {v} \cup \bigcup_{i=1}^n FV(t_i))</td>
</tr>
<tr>
<td>\langle h_1, \ldots, h_n \rangle</td>
<td>(\bigcup_{i=1}^n FV(h_i))</td>
</tr>
<tr>
<td>tv_i(h_1, \ldots, h_n)</td>
<td>{tv} \cup \bigcup_{i=1}^n FV(h_i)</td>
</tr>
<tr>
<td>\langle c \rangle</td>
<td>{}</td>
</tr>
<tr>
<td>\langle c : frame(tv[t_1, \ldots, t_m]) \rangle</td>
<td>{tv} \cup \bigcup_{i=1}^n FV(t_i)</td>
</tr>
</tbody>
</table>

Figure 3.12: Store free variables

<table>
<thead>
<tr>
<th>d</th>
<th>FV(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tv: k</td>
<td>{}</td>
</tr>
<tr>
<td>v: t</td>
<td>FV(t)</td>
</tr>
<tr>
<td>l: t</td>
<td>FV(t)</td>
</tr>
<tr>
<td>tv: k = typedef</td>
<td>FV(typedef)</td>
</tr>
<tr>
<td>v: t = b</td>
<td>FV(t) \cup FV(b)</td>
</tr>
<tr>
<td>l: t = h</td>
<td>FV(t) \cup FV(h)</td>
</tr>
</tbody>
</table>

Figure 3.13: Definition free variables

3.5.3 Type substitution

Substitution typedef[u/\alpha] is defined over type definitions, substituting a type u for the type variable \alpha in type definition typedef. Throughout this document, we assume that binding variables are renamed as necessary to avoid capture. Type substitution is shown in Figure 3.14.

Type substitution for type definitions is shown in Figure 3.15.
\begin{align*}
t & \rightarrow t[u/\alpha] \\
\mathbb{Z}_{31} & \rightarrow \mathbb{Z}_{31} \\
\text{enum}_i & \rightarrow \text{enum}_i \\
\mathbb{Z}_{\text{sign}} & \rightarrow \mathbb{Z}_{\text{sign}} \\
\mathbb{R}_{\text{pre}} & \rightarrow \mathbb{R}_{\text{pre}} \\
\langle t_1, \ldots, t_n \rangle_{\text{tuple\_class}} & \rightarrow \langle t_1[u/\alpha], \ldots, t_n[u/\alpha] \rangle_{\text{tuple\_class}} \\
\text{array} & \rightarrow t[u/\alpha] \text{ array} \\
\text{union}(tv[t_1, \ldots, t_m], \text{set}_1) & \rightarrow \text{union}(tv[t_1[u/\alpha], \ldots, t_m[u/\alpha]], \text{set}_1) \\
\text{data} & \rightarrow \text{data} \\
\text{frame}(tv[t_1, \ldots, t_m]) & \rightarrow \text{frame}(tv[t_1[u/\alpha], \ldots, t_m[u/\alpha]]) \\
(t_1, \ldots, t_n) & \rightarrow t[u/\alpha] \\
\beta &\left\{ \begin{array}{ll} u & \text{if } \beta = \alpha \\ \beta & \text{otherwise} \end{array} \right. \\
tv[t_1, \ldots, t_m] & \rightarrow tv[t_1[u/\alpha], \ldots, t_m[u/\alpha]] \\
\forall \beta_1, \ldots, \beta_m \cdot t & \left\{ \begin{array}{ll} \forall \beta_1, \ldots, \beta_m \cdot t & \text{if } \alpha \in \{\beta_1, \ldots, \beta_m\} \\ \forall \beta_1, \ldots, \beta_m \cdot t[u/\alpha] & \text{otherwise} \end{array} \right. \\
\exists \beta_1, \ldots, \beta_m \cdot t & \left\{ \begin{array}{ll} \exists \beta_1, \ldots, \beta_m \cdot t & \text{if } \alpha \in \{\beta_1, \ldots, \beta_m\} \\ \exists \beta_1, \ldots, \beta_m \cdot t[u/\alpha] & \text{otherwise} \end{array} \right. \\
v. i & \rightarrow v. i \\
\end{align*}

Figure 3.14: Type substitution

\begin{align*}
t & \rightarrow t[u/\alpha] \\
\langle t_1, \ldots, t_n \rangle & \rightarrow \langle t_1[u/\alpha], \ldots, t_n[u/\alpha] \rangle \\
t_1 \ldots t_n & \rightarrow t_1[u/\alpha] + \cdots + t_n[u/\alpha] \\
\{t_1^{\dagger} : t_1, \ldots, t_n^{\dagger} : t_n\} & \rightarrow \{t_1^{\dagger} : t_1[u/\alpha], \ldots, t_n^{\dagger} : t_n[u/\alpha]\} \\
\{t_1^{\dagger} : R_1, \ldots, t_n^{\dagger} : R_n\} & \rightarrow \{t_1^{\dagger} : R_1[u/\alpha], \ldots, t_n^{\dagger} : R_n[u/\alpha]\} \\
\Lambda \beta_1, \ldots, \beta_m \cdot \text{typedef} & \left\{ \begin{array}{ll} \Lambda \beta_1, \ldots, \beta_m \cdot \text{typedef} & \text{if } \alpha \in \{\beta_1, \ldots, \beta_m\} \\ \Lambda \beta_1, \ldots, \beta_m \cdot \text{typedef} [u/\alpha] & \text{otherwise} \end{array} \right. \\
\end{align*}

Figure 3.15: Type definition substitution
Chapter 4

Typing Rules

The typing rules for FIR terms are presented in this section as a set of inference rules. Only syntactically valid terms are covered by these rules. Each rule consists of a judgment from Figure 3.5 as the conclusion, and a list of judgments and other “side-conditions” (such as $i \in \mathbb{Z}_{\geq 1}$) as premises.

Recall that we use the notation $[v_i]_1^n$ to denote the vector $v_1, \ldots, v_n$. The index variable is implicitly $i$. An alternate notation $[v_j]_{m=1}^n$ explicitly uses a different index variable $j$. We also use the notation $[u_i/c_i]_1^n$ to represent a vector of type substitutions.

We use the shorthand $\Gamma \vdash [J_i]_1^n$ to denote a list of judgments. This is equivalent to listing all judgments $\Gamma \vdash J_1, \ldots, \Gamma \vdash J_n$. When $n = 0$, this notation expands to no judgments. The notation $[\Gamma, J_i]_1^n$ is similar.

We use the notation $(\Gamma_1, \Gamma_2)$ as $\Gamma$ to indicate that the context $\Gamma$ is the same as $(\Gamma_1, \Gamma_2)$ throughout an inference or reduction rule.

4.1 Context well-formedness

The WF-EMPTY-CONTEXT rule is the axiom rule for all type proofs and says that the empty context is well-formed.

$$\Gamma \vdash \emptyset$$

WF-EMPTY-CONTEXT

The next three rules are for proving arbitrary judgments $J$ by thinning declarations from the context. We assume implicitly that sequents are closed. For example, the THIN-TYPE rule can only be applied if $tv$ is not free in $\Gamma$.

$$\Gamma \vdash J \quad \Gamma, tv : k \vdash J$$

THIN-TYPE

$$\Gamma \vdash J \quad \Gamma \vdash t : \Omega$$

$$\Gamma, v : t \vdash J$$

THIN-VAR

$$\Gamma \vdash J \quad \Gamma \vdash t : \Omega$$

$$\Gamma, t : t \vdash J$$

THIN-GLOBAL
The following three rules are for proving arbitrary judgments \( J \) from context definitions. The judgment \( J \) is true if it can be proved with only the declaration, and if the value being defined is well-typed.

\[
\begin{align*}
\Gamma, tv : k \vdash \text{typedef} : k & \quad \Gamma, tv : k \vdash J \\
\frac{\Gamma, tv : k = \text{typedef} \vdash J}{\Gamma, tv : k = \text{typedef} \vdash J} & \quad \text{THIN-TYPE-Def} \\
\Gamma, v : t \vdash b : t & \quad \Gamma, v : t \vdash J \\
\frac{\Gamma, v : t = b \vdash J}{\Gamma, v : t = b \vdash J} & \quad \text{THIN-VAR-Def} \\
\Gamma, l : t \vdash h : t & \quad \Gamma, l : t \vdash J \\
\frac{\Gamma, l : t = h \vdash J}{\Gamma, l : t = h \vdash J} & \quad \text{THIN-GLOBAL-Def}
\end{align*}
\]

### 4.2 Type equality and structural rules

Type equality is an equivalence relation. If a type constructor is declared as part of the environment, the reflexive form reduces to context well-formedness.

\[
\begin{align*}
\Gamma, tv : k \vdash \varphi & \quad \Gamma, tv : k \vdash tv = tv : k \\
\frac{\Gamma, tv : k \vdash tv = tv : k}{\Gamma, tv : k \vdash tv = tv : k} & \quad \text{TY-Ref}
\end{align*}
\]

The symmetric and transitive forms have the usual definitions.

\[
\begin{align*}
\Gamma \vdash t_2 = t_1 : k & \quad \Gamma \vdash t_1 = t_2 : k \\
\frac{\Gamma \vdash t_2 = t_1 : k}{\Gamma \vdash t_1 = t_2 : k} & \quad \text{TY-Sym}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_1 = t_2 : k & \quad \Gamma \vdash t_2 = t_3 : k \\
\frac{\Gamma \vdash t_1 = t_2 : k}{\Gamma \vdash t_1 = t_3 : k} & \quad \text{TY-Trans}
\end{align*}
\]

If two types are equal, they contain the same elements. In the following rule, the term \( \varphi \) represents an atom \( a \), an expression \( e \), a store value \( b \), or a global label \( l \).

\[
\begin{align*}
\Gamma \vdash t = u : k & \quad \Gamma \vdash \varphi : u \\
\frac{\Gamma \vdash t = u : k \quad \Gamma \vdash \varphi : u}{\Gamma \vdash \varphi : t} & \quad \text{TY-Subst}
\end{align*}
\]

### 4.3 Application and elimination type equality rules

There are two main rules for type equality of type applications. Two type applications are equal if they use the same type definition and they have equal type parameters.

\[
\begin{align*}
\Gamma \vdash [t_1^1 = t_2^1 : \omega_1^m] & \quad \Gamma \vdash tv : \omega^m \rightarrow k_s \\
\frac{\Gamma \vdash tv[t_1^1, \ldots, t_m^1] = tv[t_1^2, \ldots, t_m^2] : k_s}{\Gamma \vdash tv[t_1^1, \ldots, t_m^1] = tv[t_1^2, \ldots, t_m^2] : k_s} & \quad \text{TY-APPLY-1}
\end{align*}
\]

If the context contains a type definition, the type in a type application may be expanded.

\[
\begin{align*}
\Gamma \vdash u_1 [t_1 / \alpha_i^m] = u_2 : k_s & \quad \Gamma \vdash tv[t_1, \ldots, t_m] : k_s \\
\frac{\Gamma', tv : k = \Lambda \alpha_1, \ldots, \alpha_m, u_1) \text{ as } \Gamma \vdash tv[t_1, \ldots, t_m] = u_2 : k_s}{\Gamma', tv : k = \Lambda \alpha_1, \ldots, \alpha_m, u_1) \text{ as } \Gamma \vdash tv[t_1, \ldots, t_m] = u_2 : k_s} & \quad \text{TY-APPLY-2}
\end{align*}
\]

The type projection \( v.i \) represents a type argument packed in a value with existential type. If \( v \) is a value of type \( \exists \alpha_0, \ldots, \alpha_{m-1}, t' \), then \( v.i \) represents the type \( \alpha_i \). If the value is defined as \( \text{ty_pack}[u](v', u_0, \ldots, u_{m-1}) \),
then the type $v \cdot i$ can be reduced to $u_i$.

$$i \in \{0 \ldots m-1\} \quad \Gamma \vdash u_i = t : \omega$$

$$(\Gamma', v : \exists \alpha_0, \ldots, \alpha_{m-1}, t'^{=}\typack[u](t', u_0, \ldots, u_{m-1})); \text{as} \quad \Gamma \vdash v \cdot i = t : \omega$$

**TY-PROJECT-EQUAL**

### 4.4 Constructor equality

The following five rules present the type equality judgments for FIR type constructors.

Two type lists are equal if they are pointwise equal. The premise $\Gamma \vdash \phi$ is required in WF-TYPELIST since it can be the case that $n = 0$, i.e. the type lists are empty.

$$\Gamma \vdash [t_1^{\phi} = t_2^{\phi} : \omega]_1^n \quad \Gamma \vdash \phi$$

$$(\Gamma \vdash \langle t_1^{\phi}, \ldots, t_n^{\phi} \rangle = \langle t_1^{\phi}, \ldots, t_n^{\phi} \rangle : \omega)$$

**WF-TYPELIST**

By default, type definitions are parameterized: the case $m = 0$ corresponds to a non-parameterized definition. Two definitions are equal if the resulting types are equal when instantiated with the same types. Note that we use implicit alpha renaming in this rule to require that the two types have the same binding variables.

$$\Gamma, [\alpha_i : \omega]_1^n \vdash \text{typedef}_s = \text{typedef}_s : k_s$$

$$(\Gamma \vdash \Lambda \alpha_1, \ldots, \alpha_m. \text{typedef}_s = \Lambda \alpha_1, \ldots, \alpha_m. \text{typedef}_s : \omega^m \rightarrow k_s)$$

**WF-TYDEF**

The next three rules allow progress to be made on proving the premise of WF-TYDEF. They cover the cases in which typedef is a union or frame type. All three rules require the context well-formedness premise $\Gamma \vdash \phi$ since it can be the case that $n = 0$.

The union space $(t_1^{\phi} + \cdots + t_n^{\phi})$ is the disjoint union of several tuple spaces (that is, a union is like a set of tagged tuples). Two union spaces are equal if they have the same number of tuple spaces, and the tuple spaces are equal.

$$\Gamma \vdash [t_1^{\phi} = t_2^{\phi} : \omega]_1^n \quad \Gamma \vdash \phi$$

$$(\Gamma \vdash \langle t_1^{\phi} + \cdots + t_n^{\phi} \rangle = \langle t_1^{\phi} + \cdots + t_n^{\phi} \rangle : \omega_{\text{union}[n]})$$

**WF-TYDEFUNION**

Frames are doubly nested records. The frame type $\{t'_1 : R_1, \ldots, t'_n : R_n\}$ declares sub-records named $t'_i$ with record type $R_i$, and the record type $\{t'_1 : t_1, \ldots, t'_n : t_n\}$ declares fields named $t'_i$ with type $t_i$. Two record types are equal if they have the same number of fields with the same labels, and the corresponding labels have equal types.

$$\Gamma \vdash [R_1^{\phi} = R_2^{\phi} : \text{record}]_1^n \quad \Gamma \vdash \phi$$

$$(\Gamma \vdash \{t'_1 : R_1, \ldots, t'_n : R_n\} = \{t'_1 : R_2, \ldots, t'_n : R_n\} : \omega_{\text{frame}})$$

**WF-TYDEFFRAME**

$$\Gamma \vdash [t_1^{\phi} = t_2^{\phi} : \Omega]_1^n \quad \Gamma \vdash \phi$$

$$(\Gamma \vdash \{t_1 : t_1, \ldots, t_n : t_n\} = \{t'_1 : t'_1, \ldots, t'_n : t'_n\} : \text{record})$$

**WF-TYDERECORD**

---

1 The cases for a general type $t$ are covered in Section 4.6.

2 The use of "tuple" in this context should not be confused with the tuple type $(t_1, \ldots, t_n)_{\text{tuple\_class}}$.
4.5 Store typing rules

In conjunction with the atom typing rules (Section 4.7), the following rules specify the typing rules for store values \( b \). They allow progress to be made on the premise of THIN-VAR-DEF.

Arrays are well-typed if each element has the same type and the type of the array is well-formed. The rule for tuples is similar except that the types of the elements may differ.

\[
\Gamma \vdash [a_i : t_i]^n \quad \Gamma \vdash \text{array} : \Omega \\
\Gamma \vdash \langle a_1, \ldots, a_n \rangle : \text{array} \tag{TY-STOREARRAY}
\]

\[
\Gamma \vdash \langle a_1, \ldots, a_n \rangle \subseteq \{ t_1, \ldots, t_n \} : \Omega \\
\Gamma \vdash \langle a_1, \ldots, a_n \rangle \subseteq \{ t_1, \ldots, t_n \} \text{tuple_class} : \Omega \\
\Gamma \vdash \langle a_1, \ldots, a_n \rangle : \{ t_1, \ldots, t_n \} \text{tuple_class} \tag{TY-STORETUPLE}
\]

In general, union definitions are polymorphic; the type \( tv[t_1', \ldots, t_m'] \) represents the union, named \( tv \), instantiated at types \( t_1', \ldots, t_m' \). The type \( \text{union}(tv[t_1', \ldots, t_m'], \{ j \}) \) is case \( j \) of this union. The value \( tv_j(a_1, \ldots, a_k) \) belongs to case \( j \) of a union type \( tv[t_1', \ldots, t_m'] \) if all of the following hold:

- The type \( tv[t_1', \ldots, t_m'] \) is a valid union type \( t_1' + \cdots + t_m' \).
- The case \( t_j \) is the tuple space \( \langle u_1, \ldots, u_k \rangle \).
- For each \( i \in \{1, \ldots, k\} \), the value \( a_i \) has type \( u_i \).

\[
\Gamma, [a_i : \omega = t_i']_i^m \vdash [a_i : u_i]^k \\
\Gamma \vdash \text{union}(tv[t_1', \ldots, t_m'], \{ j \}) : \Omega \\
\Gamma \vdash tv[t_1', \ldots, t_m'] = t_1' + \cdots + t_{j-1} + \langle u_1, \ldots, u_k \rangle + t_{j+1} + \cdots + t_{n-1} : \omega_{\text{union}\rangle n} \\
\Gamma \vdash tv_j(a_1, \ldots, a_k) : \text{union}(tv[t_1', \ldots, t_m'], \{ j \}) \tag{TY-STOREUNION}
\]

Functions are also polymorphic by default. Each function \( \Lambda a_1, \ldots, a_m. \lambda v_1, \ldots, v_n. c \) has type parameters \( a_1, \ldots, a_m \), actual parameters \( v_1, \ldots, v_n \), and body \( c \). The function has type \( \forall a_1, \ldots, a_m.(u_1, \ldots, u_n) \rightarrow t \) if, given arbitrary types \( a_1, \ldots, a_m \), and values \( v_1 : u_1, \ldots, v_n : u_n \), the body \( c \) has type \( t \).

\[
\Gamma, [a_i : \omega = a_i']_i^m + e : t \\
\Gamma \vdash \lambda v_1, \ldots, v_n. c : (u_1, \ldots, u_n) \rightarrow t \tag{TY-STOREFUN-MONO}
\]

\[
\Gamma, [a_i : \omega = a_i']_i^m + e : \lambda v_1, \ldots, v_n. c : (u_1, \ldots, u_n) \rightarrow t \\
\Gamma \vdash \Lambda a_1, \ldots, a_m. \lambda v_1, \ldots, v_n. : \forall a_1, \ldots, a_m.(u_1, \ldots, u_n) \rightarrow t \tag{TY-STOREFUN-POLY}
\]

Rawdata values are abstract. A value \( \langle c \rangle \) always has type \( \text{data} \).

\[
\Gamma \vdash \phi \\
\Gamma \vdash \langle c \rangle : \text{data} \tag{TY-STOREDATA}
\]

Frames are similar to rawdata values. The value \( \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle \) is an abstract value of type \( \text{frame}(tv[t_1, \ldots, t_m]) \) if the frame type is well-formed.

\[
\Gamma \vdash tv[t_1, \ldots, t_m] : \omega_{\text{frame}} \\
\Gamma \vdash \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle : \text{frame}(tv[t_1, \ldots, t_m]) \tag{TY-STOREFRAME}
\]

26
4.6 Type equality

Type well-formedness is ensured using the type equality judgments \( \Gamma \vdash t_1 = t_2 : \Omega \) and \( \Gamma \vdash t_1 = t_2 : \omega \). Recall that \( \omega \) is used to denote types that are small. Since the WF-SMALL-TYPE rule allows any \( \omega \) type to be used as a normal type, equality judgments are given for the \( \omega \) kind whenever possible.

\[
\frac{\Gamma \vdash t_1 = t_2 : \omega}{\Gamma \vdash t_1 = t_2 : \Omega} \quad \text{WF-SMALL-TYPE}
\]

Enumeration types are restricted to a maximum size of \( \text{max_enum} \), which is a predefined small positive integer.

\[
\frac{i \in \{0, \ldots, \text{max_enum} - 1\}}{\Gamma \vdash \text{enum}_i = \text{enum}_i : \omega} \quad \text{WF-ST-TyEnum}
\]

Type equality judgments for numeric FIR types are straightforward. Note that \( \mathbb{Z}_{\text{pre}} \) and \( \mathbb{R}_{\text{pre}} \) cannot be used as \( \omega \) types.

\[
\frac{\Gamma \vdash \phi}{\Gamma \vdash \mathbb{Z}_{\text{31}} = \mathbb{Z}_{\text{31}} : \omega} \quad \text{WF-ST-TyInt}
\]

\[
\frac{\Gamma \vdash \phi}{\Gamma \vdash \mathbb{Z}_{\text{pre}} = \mathbb{Z}_{\text{pre}} : \Omega} \quad \text{WF-TyRawInt}
\]

\[
\frac{\Gamma \vdash \phi}{\Gamma \vdash \mathbb{R}_{\text{pre}} = \mathbb{R}_{\text{pre}} : \Omega} \quad \text{WF-TyFloat}
\]

For the type \( \text{union}(tv[t_1, \ldots, t_m], \text{set}_t) \), \( \text{set}_t \) denotes a subset of the cases of the (polymorphic) union type named by \( tv \) instantiated at types \( t_1, \ldots, t_m \). For two union types to be equal, they must have the same name, and their type arguments must also be equal.

\[
\frac{\text{set}_t \subseteq \{0, \ldots, n - 1\}}{\Gamma \vdash \text{union}(tv[t_1, \ldots, t_m], \text{set}_t) = \text{union}(tv[t_1, \ldots, t_m], \text{set}_t) : \omega} \quad \text{WF-ST-TyUnion}
\]

Two tuple types are equal if they are pointwise equal (their projections have the same types). The context well-formedness premise \( \Gamma \vdash \phi \) is required in WF-ST-TyTuple-Normal since it can be the case that \( n = 0 \).

\[
\frac{\Gamma \vdash \left[t^1_i = t^2_i : \omega^1\right]^n}{\Gamma \vdash \left\langle t^1_1, \ldots, t^1_n \right\rangle_{\text{normal}} = \left\langle t^2_1, \ldots, t^2_n \right\rangle_{\text{normal}} : \omega} \quad \text{WF-ST-TyTuple-Normal}
\]

The \text{box} tuples are used to “pack” non-\( \omega \) types in a form that has kind \( \omega \). The tuple must have arity exactly one.

\[
\frac{\Gamma \vdash t_1 = t_2 : \Omega}{\Gamma \vdash \left\langle t_1 \right\rangle_{\text{box}} = \left\langle t_2 \right\rangle_{\text{box}} : \omega} \quad \text{WF-ST-TyTuple-Boxed}
\]

An array is just like a tuple, except that all the elements must have the same type, and the arity is not defined by the type. Two array types are equal if their element types are equal.

---

3Enumeration values are restricted to small values to reduce the number of cases where a numerical value is mistaken as a pointer index during garbage collection; all pointer indices must be larger than \( \text{max_enum} \). This reduces the likelihood that a dead block will be marked live by the garbage collector.
\[ \Gamma \vdash t_1 = t_2 : \Omega \]
\[ \Gamma \vdash t_1 \text{ array} = t_2 \text{ array} : \omega \]

WF-ST-TyArray

"Rawdata" is the opaque data space used to represent C heap data.

\[ \Gamma \vdash \phi \]
\[ \Gamma \vdash \text{data} = \text{data} : \omega \]

WF-ST-TyRawData

For two frame types to be equal, they must have the same frame name, and their type arguments must also be equal.

\[ \Gamma \vdash [t_1^1 = t_2^2 : \omega ]^m \quad \Gamma \vdash tv[t_1^1, \ldots , t_m^m] : \omega_{frame} \]
\[ \Gamma \vdash \text{frame}(tv[t_1^1, \ldots , t_m^m]) = \text{frame}(tv[t_1^1, \ldots , t_m^m]) : \omega \]

WF-ST-TyFrame

Two existential types are equal if they have the same arity, and their abstracted types are equal. Note that we use implicit alpha renaming in this rule to require that the two types have the same binding variables.

\[ \Gamma, \left[ \alpha_i : \omega \right]^m \vdash t_1 = t_2 : \Omega \]
\[ \Gamma \vdash \exists \alpha_1, \ldots , \alpha_m. t_1 = \exists \alpha_1, \ldots , \alpha_m. t_2 : \omega \]

WF-ST-TyExists

A type projection \( v. i \) is well-formed if \( v \) has existential type \( \exists \alpha_0, \ldots , \alpha_{m-1}. t \), and \( i \) is a valid index for one of the type parameters \( \alpha_i \).

\[ i \in \{0 \ldots m - 1\} \quad \Gamma \vdash v : \exists \alpha_0, \ldots , \alpha_{m-1}. t \]
\[ \Gamma \vdash v. i = v. i : \omega \]

WF-ST-TyProject

The only values that are universally polymorphic are functions, corresponding to value restriction [19]. We enforce the value restriction explicitly in the well-formedness rule for \( \forall \alpha_1, \ldots , \alpha_m. t \) by requiring \( t \) to be a function type. Again, we use implicit alpha renaming to require that the two types use the same binding sequence.

\[ \Gamma \vdash \left[ t_1^1 = t_2^2 : \Omega \right]^n \quad \Gamma \vdash u^1 = u^2 : \Omega \]
\[ \Gamma \vdash ((t_1^1, \ldots , t_n^1) \rightarrow u^1) = ((t_2^1, \ldots , t_n^2) \rightarrow u^2) : \omega \]

WF-ST-TyFun-Mono

\[ \Gamma, \left[ \alpha_i : \omega \right]^m \vdash ((t_1^1, \ldots , t_n^1) \rightarrow u^1) = ((t_2^1, \ldots , t_n^2) \rightarrow u^2) : \omega \]
\[ \Gamma \vdash \forall \alpha_1, \ldots , \alpha_m. ((t_1^1, \ldots , t_n^1) \rightarrow u^1) = \forall \alpha_1, \ldots , \alpha_m. ((t_2^1, \ldots , t_n^2) \rightarrow u^2) : \omega \]

WF-ST-TyFun-Poly

### 4.7 Atom typing rules

Atoms represent the values that are used in expressions. This section defines the type judgments for atoms.

A variable \( v \) has type \( t \) only if it is defined with type \( t \) in the context.

\[ \Gamma, v : t \vdash \phi \]
\[ \Gamma, v : t \vdash v : t \]

TY-AtomVar

Enumerated values represent small, nonnegative numbers. The value \texttt{enum}_i(j) belongs to an enumeration type \texttt{enum}_i, which represents the values \{0 \ldots i - 1\}. Note that this rule can never be applied for \texttt{enum}_0.
4.7. ATOM TYPING RULES

\[ j \in \{0, \ldots, i-1\} \quad \Gamma \vdash \text{enum}_i : \Omega \]
\[ \Gamma \vdash \text{enum}_i(j) : \text{enum}_i \quad \text{TY-AtomEnum} \]

There are numeric constants for integers, raw integers (integers with various precision and signedness interpretations), and floating-point values. The side conditions \( i \in \mathbb{Z}_{\geq 1}, \ r \in \mathbb{Z}_{\text{sign}}^{\text{pre}}, \) and \( x \in \mathbb{R}_{\text{pre}} \) are necessary since syntax alone does not ensure that these values will be in the type.

\[ i \in \mathbb{Z}_{\geq 1} \quad \Gamma \vdash \ \text{int}(i) : \mathbb{Z}_{\geq 1} \quad \text{TY-AtomInt} \]

\[ r \in \mathbb{Z}_{\text{sign}}^{\text{pre}} \quad \Gamma \vdash \ \text{rawint}_{\text{pre}}(r) : \mathbb{Z}_{\text{sign}}^{\text{pre}} \quad \text{TY-AtomRawInt} \]

\[ x \in \mathbb{R}_{\text{pre}} \quad \Gamma \vdash \ \text{float}_{\text{pre}}(x) : \mathbb{R}_{\text{pre}} \quad \text{TY-AtomFloat} \]

Labels are used to index frame objects. Frame objects are viewed as “unsafe” data areas, and may be embedded with rawdata areas. For this reason, labels are treated as signed constant integers, and may be used as numbers in arithmetic.

\[ r_{\text{off}} \in \mathbb{Z}_{\text{signed}}^{\text{pre}} \quad \Gamma \vdash \text{tv}[u_1, \ldots, u_m] = \{l^l : \{l^r : t_1, \ldots, t_n\} : \omega_{\text{frame}} \}
\[ \Gamma \vdash \text{label}(\text{tv}, l^l, l^r, r_{\text{off}}) : \mathbb{Z}_{\text{signed}}^{\text{pre}} \quad \text{TY-AtomLabel} \]

The sizeof construction has purpose similar to labels; the value \( \text{sizeof}(t v_1, \ldots, t v_n, r_{\text{size}}) \) represents the sum of the (byte) sizes of the frames defined by \( t v_1, \ldots, t v_n \) and the constant integer \( r_{\text{size}} \). The size itself is determined by the runtime environment. The context well-formedness judgment is needed since \( n \) may be 0.

\[ r_{\text{size}} \in \mathbb{Z}_{\text{signed}}^{\text{pre}} \quad \Gamma \vdash [t v_i : \omega_{\text{pre}}^m \rightarrow \omega_{\text{frame}}]_n \quad \Gamma \vdash \ \text{sizeof}(t v_1, \ldots, t v_n, r_{\text{size}}) : \mathbb{Z}_{\text{signed}}^{\text{pre}} \quad \text{TY-AtomSizeof} \]

Constant atoms correspond to zero-arity elements of unions. The atom \( \text{const}[u](t v, i) \) belongs to a union type \( \text{union}(t v[t_1, \ldots, t_n], \{i\}) \) if the \( i \)-th case of the union has zero arity. Note that the rule \( \text{TY-Apply-2} \) requires that the types \( t_1, \ldots, t_n \) be well formed types in \( \omega \). Also note that the rule implicitly requires \( i \in \{0, \ldots, n-1\} \). Thus, the union type must be well formed in \( \text{TY-AtomConst} \).

The constant atom is an optimization; zero-arity union cases can also be allocated using the allocation construction \( \text{union}(t v[t_1, \ldots, t_n], i) \) (and the corresponding type rule \( \text{TY-Alloc-Union} \)). However, the runtime environment normally pre-allocates values for these constant constructors, and the atom is a convenient reference (although the allocation has the same semantics).

\[ \Gamma \vdash u = \text{union}(t v[t_1, \ldots, t_n], \{i\}) : \omega \]
\[ \Gamma \vdash \text{tv} = t_0 + \cdots + t_{i-1} + t_0 + \cdots + t_0 + \cdots + t_{n-1} : \omega_{\text{union}[n]} \]
\[ \Gamma \vdash \text{const}[u](t v, i) : \text{union}(t v[t_1, \ldots, t_n], \{i\}) \quad \text{TY-AtomConst} \]

The following three rules represent polymorphic type operations. These three rules use type substitution defined in Section 3.5.3.

If \( a_f \) is a polymorphic function with type \( \forall \alpha_1, \ldots, \alpha_m, t \), the type application \( \text{ty} \text{apply}[u](a_f, u_1, \ldots, u_m) \) represents the function instantiated with type arguments \( u_1, \ldots, u_m \).
A type \textit{pack} construction is the existential introduction form. It packages a value with a list of types to form a value with existential type. The type $t$ is a type with (potentially) free type variables $\alpha_1, \ldots, \alpha_m$. If a value $v$ has type $t$ with types $t_i$ substituted for the variables $\alpha_i$, then $\text{ty} \cdot \text{pack}[\exists \alpha_1, \ldots, \alpha_m.t](v, t_1, \ldots, t_m)$ has type $\exists \alpha_1, \ldots, \alpha_m.t$.

\[
\Gamma \vdash v : \exists \alpha_1, \ldots, \alpha_m.t \\
\Gamma \vdash \text{ty} \cdot \text{pack}(v) : [v_i / \alpha_i]_1^{m-1}
\]

The \textit{unpack} construction is the corresponding elimination form. The unpacking does not introduce new type variables directly; rather, the types that were packed during existential introduction are referred to by their type projections. If $v$ has existential type $\exists \alpha_0, \ldots, \alpha_{m-1}.t$, then $\text{ty} \cdot \text{unpack}(v)$ has type $t$ with the type projections $v_i$ substituted for $\alpha_i$. The premise of $\text{TY-ATOM-TYUNPACK}$ ensures that the type projections $v_i$ are well-formed.

\[
\Gamma \vdash a : \text{arg}(unop) \\
\Gamma \vdash \text{res}(unop) : \Omega \\
\Gamma \vdash \text{unop} a : \text{res}(unop)
\]

The final atom values correspond to arithmetic. The unary rule uses the unary syntax table defined in Section 6. The $\text{res}(unop)$ is meta-notation that refers to the result type of the unary operator, and $\text{arg}(unop)$ is the type of the argument.

\[
\Gamma \vdash \text{arg}_1(binop) \\
\Gamma \vdash \text{arg}_2(binop) \\
\Gamma \vdash \text{res}(binop) : \Omega \\
\Gamma \vdash a_1 \text{ binop } a_2 : \text{res}(binop)
\]

4.8 Expression typing

Expressions form function bodies, and are the primary form used for program evaluation.

4.8.1 Basic expression typing rules

Operationally, the $\text{let } v : t_1 = a \text{ in } e$ expression evaluates the atom $a$, binds the variable $v$ to the value, and then evaluates expression $e$. The expression has type $t_2$ if the atom has type $t_1$, and the expression $e$ has type $t_2$ when $v$ has type $t_1$.

\[
\Gamma \vdash a : t_1 \\
\Gamma, v : t_1 \vdash e : t_2 \\
\Gamma \vdash \text{let } v : t_1 = a \text{ in } e : t_2
\]

External forms are defined by the runtime environment to access the runtime or the operating system. For example, the runtime typically exports functions to gather statistics and control the garbage collector, and
4.8. EXPRESSION TYPING

it also exports a set of system calls. The description of these functions is specific to the runtime, and we omit discussion in this paper. For our purposes, we assume there is some interpretation \([“s”]\) that defines a function that corresponds to the runtime operation.

\[
\Gamma \vdash [a_i : u_i^n]_1 \quad \Gamma, v : t_1 + e : t_2 \quad \Gamma \vdash [“s”] : (u_1, \ldots, u_n) \to t_1 \\
\Gamma \vdash \text{let } v : t_1 = (“s” : (u_1, \ldots, u_n) \to t_1)(a_1, \ldots, a_n) \text{ in } e : t_2
ty\text{-LetExt}
\]

Tail-calls represent function calls. This rule assumes that FIR programs use continuation-passing-style\(^4\). In a tail-call \(a_f(a_1, \ldots, a_n)\), the atom \(a_f\) refers to a function that is called with arguments \(a_1, \ldots, a_n\). Functions are higher-order, and may be passed as arguments to other functions, or stored in arrays, tuples, or other values. The atom \(a_f\) is typically a type application \(f[t_1, \ldots, t_m]\) of some function \(f\) to type arguments \(t_1, \ldots, t_m\).

\[
\Gamma \vdash a_f : (u_1, \ldots, u_n) \to \text{enum}_0 \\
\Gamma \vdash [a_i : u_i^n]_1 \\
\Gamma \vdash a_f(a_1, \ldots, a_n) : \text{enum}_0
ty\text{-TailCall}
\]

4.8.2 Special-call typing rules

Special-calls are expressions for specifying process migration and atomic transactions. For simplicity, the expression typing rule \(ty\text{-Special-Call}\) uses a common form for all of the special-calls. If \(tailop\) is a special-call, then \(\text{special } tailop\) is an expression.

\[
\Gamma \vdash \text{tailop} : \text{special } t \\
\Gamma \vdash \text{special } tailop : t
ty\text{-Special-Call}
\]

Process migration is specified with the special-call expression \(\text{migrate } [j, a_{pdr}, a_{off}] a_f(a_1, \ldots, a_n)\), which is described in Section 8.2. The atoms \(a_{pdr}\) and \(a_{off}\) specify a string (as a raw data block and offset) that describes the migration protocol and target (for example, a machine name). Section 8.2.1 describes the possible protocols available for system migration. The \(a_f(a_1, \ldots, a_n)\) is a tail-call to be performed once the process has migrated. The number \(j\) is a unique identifier used by the runtime.

\[
\Gamma \vdash a_{pdr} : \text{data} \\
\Gamma \vdash [a_i : t_i^n]_1 \\
\Gamma \vdash a_f : (t_1, \ldots, t_n) \to \text{enum}_0 \\
\Gamma \vdash \text{migrate } [j, a_{pdr}, a_{off}] a_f(a_1, \ldots, a_n) : \text{special } \text{enum}_0
ty\text{-SysMigrate}
\]

The “atomic” special-calls are described in Section 8.3. The expression \(\text{atomic } a_f(a_{const}, a_1, \ldots, a_n)\) specifies entry into a new atomic transaction, which may be later committed or rolled back. The atomic call instructs the runtime to establish a process checkpoint that may be used for rollback (or discarded if the transaction is committed). Otherwise, the atomic call acts exactly like a tail-call to a function \(a_f\) with arguments \((a_{const}, a_1, \ldots, a_n)\). For technical reasons, the first argument is currently restricted to have type \(\mathbb{Z}_{32}^{\text{signed}}\).

\[
\Gamma \vdash a_{const} : \mathbb{Z}_{32}^{\text{signed}} \\
\Gamma \vdash [a_i : t_i^n]_1 \\
\Gamma \vdash a_f : (\mathbb{Z}_{32}^{\text{signed}}, t_1, \ldots, t_n) \to \text{enum}_0 \\
\Gamma \vdash \text{atomic } a_f(a_{const}, a_1, \ldots, a_n) : \text{special } \text{enum}_0
ty\text{-Atomic}
\]

It is possible to enter several transactions simultaneously (in the source program, transactions are typically nested). Each program checkpoint is identified by an integer \(a_{level}\)\(^5\). When a transaction at level \(a_{level}\) is aborted, or “rolled-back”, all transactions with more recent identifiers are discarded and the program state

---

\(^4\)It would be straightforward to add a returning-function expression.

\(^5\)Atomic levels are described in further detail in section 8.3.1.
is restored to the state immediately after entry into the transaction. If the transaction was initiated with 
atomic \(a'_{\text{fun}}(a'_{\text{const}}, a'_1, \ldots, a'_{n})\), the rollback \([a_{\text{level}}, a_{\text{const}}]\) special-call expression resumes execution with a tail-call to \(a'_{\text{fun}}(a_{\text{const}}, a'_1, \ldots, a'_{n})\).

\[
\Gamma \vdash a_{\text{level}} : \mathbb{Z}_{\text{sig}}^{\text{signed}} \quad \Gamma \vdash a_{\text{const}} : \mathbb{Z}_{\text{sig}}^{\text{signed}} \\
\Gamma \vdash \text{rollback} [a_{\text{level}}, a_{\text{const}}] : \text{specialenum}_0
\]

TY-AtomicRollback

Transactions can be committed in any order. The expression commit \([a_{\text{level}}], a_f(a_1, \ldots, a_n)\) specifies that 
transaction \(a_{\text{level}}\) be committed. Committing a transaction instructs the runtime to discard the program 
checkpoint identified by \(a_{\text{level}}\). Once committed, the commit call acts like the tail-call \(a_f(a_1, \ldots, a_n)\).

\[
\Gamma \vdash a_{\text{level}} : \mathbb{Z}_{\text{sig}}^{\text{signed}} \quad \Gamma \vdash a_i : t_i^n \quad \Gamma \vdash a_f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0 \\
\Gamma \vdash \text{commit} [a_{\text{level}}, a_f(a_1, \ldots, a_n)] : \text{specialemm}_0
\]

TY-AtomicCommit

### 4.8.3 Match statement typing rules

Match statements allow pattern matching on numbers, where a pattern is defined as a set of constant intervals 
(patterns do not contain variables). The type of the atom being analyzed determines the rule to apply. A
match statement with zero cases is not allowed; the type of the atom must contain at least one value. In
addition, partial match statements are not allowed; the union of the sets must cover all possible values for
the type of the atom.

The following three rules are similar. Operationally, a match expression match \(a\) with \([s_i \mapsto e_i]_1^n\) chooses the first expression \(e_i\) for evaluation where \(a \in s_i\). The match expression has type \(t\) if all of the expressions 
\(e_i\) have type \(t\) and the sets \(s_i\) cover all of the possible values for the match argument \(a\).

\[
[s_i \in \text{set}_1]_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{\text{sp}}^{\text{int}} \quad \Gamma \vdash a : \mathbb{Z}_{\text{sp}}^{\text{int}} \quad \Gamma \vdash [e_i : t]_1^n \\
\Gamma \vdash \text{match} a \text{ with } [s_i \mapsto e_i]_1^n : t
\]

TY-Match-Int

\[
j > 0 \quad [s_i \in \text{set}_1]_1^n \quad \bigcup_{i=1}^n s_i = \text{enum}_j \quad \Gamma \vdash a : \text{enum}_j \quad \Gamma \vdash [e_i : t]_1^n \\
\Gamma \vdash \text{match} a \text{ with } [s_i \mapsto e_i]_1^n : t
\]

TY-Match-Enum

\[
[s_i \in \text{set}_2]_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{\text{sp}}^{\text{pre}} \quad \Gamma \vdash a : \mathbb{Z}_{\text{sp}}^{\text{pre}} \quad \Gamma \vdash [e_i : t]_1^n \\
\Gamma \vdash \text{match} a \text{ with } [s_i \mapsto e_i]_1^n : t
\]

TY-Match-RawInt

The match case for unions is a special case. A value \(v\) of type \(\text{union}(tv[t_1, \ldots, t_m], s_{\text{union}})\) belongs to some 
union space \(t_1 + \ldots + t_m\). The actual value for \(v\) belongs to exactly one of the cases in the union, and the 
match \(v\) with \([s_i \mapsto e_i]_1^n\) expression is the elimination form. For the match statement to be well-formed, 
the atom \(v\) must be a variable with a valid union type restricted to cases \(s_{\text{union}}\). The match cases \(s_i\) must 
include all of the possible values in \(s_{\text{union}}\). For case \(s_j \mapsto e_j\), the expression \(e_j\) must be well-typed given the 
restricted type \(\text{union}(tv[t_1, \ldots, t_m], s_j)\) for \(v\).

\[
[s_i \in \text{set}_1]_1^n \quad \{s_i \neq s_{\text{union}}\} \quad \bigcup_{i=1}^n s_i = \bigcup_{i=1}^n s_i \quad \Gamma, v : \text{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t \quad [s_i \mapsto e_i]_1^n \\
\Gamma, v : \text{union}(tv[t_1, \ldots, t_m], s_{\text{union}}) \vdash \text{match} v \text{ with } [s_i \mapsto e_i]_1^n : t
\]

TY-Match-Union

32
4.8.4 Allocation typing rules

For memory operations, we define two short-hand macros that specify information on the offset atoms. Some operations permit an integer or a variable offset, which must have type \( \mathbb{Z}_{31} \) or \( \mathbb{Z}_{32}^{\text{signed}} \); this alternation is expressed with the offset macro. Other operations restrict the offset atom to be a constant; these use the offset\(^{\text{const}}\)\((i)\) macro, where \( c \) is a tuple class and \( i \) is a word offset into a tuple. Note that word\(_{\text{size}}\) is a constant, and indicates the size of a word on the architecture.

\[
\text{offset} = \mathbb{Z}_{31} | \mathbb{Z}_{32}^{\text{signed}}
\]

\[
\text{offset}^{\text{const}}(i) = \begin{cases} 
\text{int}(i) & | \text{rawint}^{\text{signed}}_{32}(i \cdot \text{word\_size}) & \text{when } c = \text{normal} \\
\text{int}(0) & | \text{rawint}^{\text{signed}}_{32}(0) & \text{when } c = \text{box}
\end{cases}
\]

For simplicity, the allocation typing rule uses a common form for all of the allocation operators. If alloc is an allocation of type alloc\(\ u\), and \( e \) is well-typed under the assumption that \( v \) has type \( u \), then let \( v = \text{alloc in} \ e : t \) is a well-typed allocation expression.

\[
\frac{\Gamma \vdash \text{alloc} \ : \ u \quad \Gamma, v : u \vdash e \ : t}{\Gamma \vdash \text{let} v = \text{alloc in} \ e : t} \quad \text{TY-ALLOC}
\]

There are five allocation operations, corresponding to the five different kinds of aggregate values: tuples, arrays, unions, rawdata, and frames\(^6\).

Tuples are allocated with \( \langle a_1, \ldots, a_n \rangle_c \), where \( a_1, \ldots, a_n \) are the initial values in the tuple, and \( c \) is the tuple class.

\[
\frac{\Gamma \vdash [a_i : u_i]^n \quad \Gamma \vdash \langle u_1, \ldots, u_n \rangle_c : \omega}{\Gamma \vdash \langle a_1, \ldots, a_n \rangle_c : \text{alloc} \langle u_1, \ldots, u_n \rangle_c} \quad \text{TY-ALLOC-TUPLE}
\]

Arrays are allocated by specifying the number \( a_{\text{size}} \) of elements in the array, and the initial value \( a_{\text{init}} \) for all the elements in the array.

\[
\frac{\Gamma \vdash a_{\text{size}} : \text{offset} \quad \Gamma \vdash a_{\text{init}} : u}{\Gamma \vdash \text{array} (a_{\text{size}}, a_{\text{init}}) : \text{alloc} \ u \ \text{array}} \quad \text{TY-ALLOC-ARRAY}
\]

Unions are allocated with the form union\(\ tv[a_1, \ldots, a_k], j \), where \( tv \) must be a union; union case \( j \) must be a tuple \( \langle u_1, \ldots, u_k \rangle \) with arity \( k \), and value \( a_i \) must have type \( u_i \). The rule here defers to the Ty-StoreUnion to spell out these requirements—the allocation is well-formed if the resulting store value is well-formed.

\[
\frac{\Gamma \vdash tv[j(a_1, \ldots, a_k) : \text{union}(tv[t_1, \ldots, t_m], \{j\})]}{\Gamma \vdash \text{union}(tv[a_1, \ldots, a_k], j) : \text{allocunion}(tv[t_1, \ldots, t_m], \{j\})} \quad \text{TY-ALLOC-UNION}
\]

The malloc form malloc\(\ a \) is used to allocate rawdata blocks. The atom \( a \) is the minimum number of bytes in the rawdata area.

\[
\frac{\Gamma \vdash a : \text{offset}}{\Gamma \vdash \text{malloc}(a) : \text{alloc data}} \quad \text{TY-ALLOC-MALLOC}
\]

Frame allocation using the form frame\(\ tv[u_1, \ldots, u_m] \) is similar to rawdata allocation, in that it doesn’t

\(^6\)For readability, we shorten \( [a_1, \ldots, a_n]_c : t \) to \( [a_1, \ldots, a_n]_c \) in this section. The same is done for the union, frame, and rawdata allocation operators. The type \( t \) in these operators (that is dropped) is the same as the type given to the operator by the typing rules.
specify the initial values in the frame. The allocation is well-formed if the type \( \text{frame}(tv[u_1, \ldots, u_m]) \) is well-formed. The number of bytes allocated is at least \( \text{sizeof}(tv, 1) \).

\[
\Gamma \vdash \text{frame}(tv[u_1, \ldots, u_m]) : \omega \\
\Gamma \vdash \text{frame}(tv[u_1, \ldots, u_m]) : \text{allocframe}(tv[u_1, \ldots, u_m])
\]

**Ty-Alloc-Frame**

### 4.8.5 Subscripting typing rules

Elements in the aggregate types (tuples, arrays, unions, rawdata, and frames) are accessed using the `let v: t_1 = a_1[a_2] in e` expression. All entries in aggregate blocks are mutable; an entry is replaced using the `a_1[a_2]: t_1 \leftarrow a_3; e` expression.

In general, `let v: t_1 = a_1[a_2] in e` is a well-formed expression if all of the following are true.

- The atom \( a_1 \) has an aggregate type.
- The atom \( a_2 \) is a valid index into \( a_1 \).
- The element of \( a_1 \) at location \( a_2 \) has type \( t_1 \).
- Expression \( e \) is well-formed assuming \( v \) has type \( t_1 \).

Similarly, \( a_1[a_2]: t_1 \leftarrow a_3; e \) is well-formed if the following are true.

- The atom \( a_1 \) has aggregate type.
- The atom \( a_2 \) is a valid index into \( a_1 \).
- The element of \( a_1 \) at location \( a_2 \) has type \( t_1 \).
- The atom \( a_3 \) has type \( t_1 \).
- Expression \( e \) is well-formed (with no extra assumptions).

For the subscripting operation \( a_1[a_2] \) on tuple aggregates, the index \( a_2 \) must be a constant \( j \). If the tuple \( a_1 \) has type \( \langle u_0, \ldots, u_{n-1} \rangle \), the element has type \( u_j \).

\[
\Gamma \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_e \\
\Gamma, v_1 : u \vdash e : t \\
\Gamma \vdash \text{let } v_1 : u = a_1[\text{offset}^\text{const}_e(j)] \text{ in } e : t
\]

**Ty-LetSub-Tuple**

\[
\Gamma \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_e \\
\Gamma \vdash a_3 : u \\
\Gamma \vdash e : t \\
\Gamma \vdash a_1[\text{offset}^\text{const}_e(j)] : u \leftarrow a_3; e : t
\]

**Ty-SetSub-Tuple**

Unlike tuples, all the elements in an array have the same type. The subscripting operation \( a_1[a_2] \) is well-formed if the index \( a_2 \) is a valid index, the aggregate \( a_1 \) has array type \( u \text{ array} \), and the element being accessed has type \( u \).

\[
\Gamma \vdash a_1 : u \text{ array} \\
\Gamma \vdash a_2 : \text{offset} \\
\Gamma, v : u \vdash e : t \\
\Gamma \vdash \text{let } v : u = a_1[a_2] \text{ in } e : t
\]

**Ty-LetSub-Array**

\[
\Gamma \vdash a_1 : u \text{ array} \\
\Gamma \vdash a_2 : \text{offset} \\
\Gamma \vdash a_3 : u \\
\Gamma \vdash e : t \\
\Gamma \vdash a_1[a_2] : u \leftarrow a_3; e : t
\]

**Ty-SetSub-Array**
Union operations are somewhat more complicated. For the subscripting operation $a_1[a_2]$ to be well-formed, the aggregate $a_1$ must belong to a specific case $\bar{u}_j$ in some union type $\bar{u}_0 + \cdots + \bar{u}_{m-1}$. The index $a_2$ must be a constant $k$. The type list $\bar{u}_j$ must be a vector $\langle u'_0, \ldots, u'_{l-1} \rangle$, and the element being accessed must have type $u'_k$.

$$
\Gamma \vdash a_1 : \text{union}(tv[t'_1, \ldots, t'_{m}], \{j\})
$$

$$
\Gamma \vdash tv[t'_1, \ldots, t'_{m}] = \left(\left[u'_{j-1}\right]_0^{-1} + \langle \left[u'_{j-1}\right]_0^{k-1}, u, \left[u'_{j-1}\right]_0^{l-1} \rangle + \left[u'_{j+1}\right]_0^{n-1} \right) : \omega_{\text{union}}[n]
$$

$$
\Gamma, v : u \vdash e : t \quad \text{TY-LetSub-Union}
$$

$$
\Gamma \vdash a_1 : \text{union}(tv[t'_1, \ldots, t'_{m}], \{j\})
$$

$$
\Gamma \vdash tv[t'_1, \ldots, t'_{m}] = \left(\left[u'_{j-1}\right]_0^{-1} + \langle \left[u'_{j-1}\right]_0^{k-1}, u, \left[u'_{j-1}\right]_0^{l-1} \rangle + \left[u'_{j+1}\right]_0^{n-1} \right) : \omega_{\text{union}}[n]
$$

$$
\Gamma \vdash a_3 : u
$$

$$
\Gamma \vdash e : t \quad \text{TY-SetSub-Union}
$$

For load/store operations on data, the type of value being accessed is not determined by the typing rules. The data type is used primarily for C programs, which allows data in memory to be manipulated with arbitrary types. For safety, the runtime environment generates a runtime safety check to verify that the value being accessed has a compatible type.

$$
\Gamma \vdash a_1 : \text{data} \quad \Gamma \vdash a_2 : \text{offset} \quad \Gamma, v : u \vdash e : t
$$

$$
\Gamma \vdash \text{let } v : a_1[a_2] \text{ in } e : t
$$

$$
\text{TY-LetSub-RawData}
$$

$$
\Gamma \vdash a_1 : \text{data} \quad \Gamma \vdash a_2 : \text{offset} \quad \Gamma \vdash a_3 : u
$$

$$
\Gamma \vdash a_1[a_2] : u \leftarrow a_3; e : t
$$

$$
\text{TY-SetSub-RawData}
$$

Frames are similar to rawdata, in that the runtime generates runtime safety checks because frames are allocated uninitialized. Otherwise, a subscript operation $a_1[a_2]$ is well formed if $a_2$ is a two-level label $\langle l', l' \rangle$, $a_1$ is a frame, and the element being accessed has the same type as the field $a_1, l', l'$.

$$
r_{off} \in \mathbb{Z}_{\text{signed}}
$$

$$
\Gamma \vdash a_1 : \text{frame}(tv[t'_1, \ldots, t'_{m}])
$$

$$
\Gamma \vdash tv[t'_1, \ldots, t'_{m}] = \{l' : \{l' : u; \cdots; \cdots\} : \omega_{\text{frame}}
$$

$$
\Gamma, v : u \vdash e : t
$$

$$
\Gamma \vdash \text{let } v : a_1[\text{label}(tv, l', l', r_{off})] \text{ in } e : t
$$

$$
\text{TY-LetSub-Frame}
$$

$$
r_{off} \in \mathbb{Z}_{\text{signed}}
$$

$$
\Gamma \vdash a_1 : \text{frame}(tv[t_1, \ldots, t_m])
$$

$$
\Gamma \vdash tv[t_1, \ldots, t_m] = \{l' : \{l' : u; \cdots; \cdots\} : \omega_{\text{frame}}
$$

$$
\Gamma \vdash a_3 : u
$$

$$
\Gamma \vdash e : t
$$

$$
\Gamma \vdash a_1[\text{label}(tv, l', l', r_{off})] : u \leftarrow a_3; e : t
$$

$$
\text{TY-SetSub-Frame}
$$

### 4.8.6 Global label typing rules

Global values are represented directly in the context using a label $l$. For global operations to be well-formed, the label $l$ must be a valid global label with type $t$. 
\[
\begin{align*}
\Gamma, l: t \vdash \phi \\
\Gamma, l: t \vdash l: t
\end{align*}
\]
\text{Ty-Label}

Labels are \textit{not} variables; there is no atom that allows global values to be used directly. The value in a global is loaded with the \texttt{let} \( v : t_1 = \texttt{global} \in e \) expression, and stored with the \texttt{global}: \( t_1 \leftarrow a; e \) expression.

\[
\begin{align*}
\Gamma \vdash l: u & \quad \Gamma, v: u \vdash e : t \\
\Gamma \vdash \texttt{let} \ v: u = \texttt{global} \in e : t
\end{align*}
\]
\text{Ty-LetGlobal}

\[
\begin{align*}
\Gamma \vdash l: u & \quad \Gamma \vdash a: u \quad \Gamma \vdash e : t \\
\Gamma \vdash \texttt{global}: u \leftarrow a; e : t
\end{align*}
\]
\text{Ty-SetGlobal}
Chapter 5

Operational Semantics

Evaluation is defined on programs, which include three parts: the current environment $\Gamma$, a checkpoint environment $C$, which is an ordered list of checkpoints, and an expression $e$ to be evaluated. Checkpoints are required for transactions, which are discussed in Section 5.4. A checkpoint $(\Gamma, f(\alpha, a_1, \ldots, a_n))$ contains a context $\Gamma$, and a function $f(\alpha, a_1, \ldots, a_n)$, where $\alpha$ is a special transaction parameter. The function is called if evaluation is resumed from the checkpoint.

$$
C ::= \langle \Gamma, f(\alpha, a_1, \ldots, a_n) \rangle \quad \text{Single checkpoint}
$$

$$
C ::= C_m; \ldots; C_1 \quad \text{Checkpoint environment}
$$

**Definition 5.0.1 Fully-defined Contexts**

A context $\Gamma$ is said to be fully-defined if every variable $v$ in the context is defined with the form $v: t = b$, every type variable $tv$ is defined with the form $tv: k = \text{typedef}$, and every global label $l$ is defined with the form $l: t = h$.

**Definition 5.0.2 Programs**

A program is either the special term error, or it is a triple $(\Gamma \mid C \mid e)$ that satisfies the following conditions.

- $\Gamma$ is fully-defined and $\Gamma \vdash e : \text{enum}_0$.
- For each $(\Gamma', f(\alpha, a_1, \ldots, a_n)) \in C$, the context $\Gamma'$ is fully-defined, and the judgment $\Gamma', v_{\text{const}}: \mathbb{Z}_{\geq 2} \vdash f(v_{\text{const}}, a_1, \ldots, a_n) : \text{enum}_0$ holds.

Intuitively, the error term specifies a runtime error during program evaluation (such as an out-of-bounds array access, or a runtime type error during a subscripting operation).

Evaluation is defined as a relation on programs. The relation

$$(\Gamma \mid C \mid e) \rightarrow (\Gamma' \mid C' \mid e')$$

specifies that the program $(\Gamma \mid C \mid e)$ evaluates in one step to the program $(\Gamma' \mid C' \mid e')$. The relation

$$(\Gamma \mid C \mid e) \rightarrow \text{error}$$
specifies that evaluation of the program \((\Gamma \mid \mathcal{C} \mid e)\) results in a runtime error in one step.

In this section, we specify program evaluation in two parts. First, we specify leftmost-innermost evaluation of the atoms in an expression. Second, we specify evaluation of expressions in which the atoms are heap values. Atom evaluation is defined using expression contexts, discussed in the following section.

To ease readability of the rules, we use informal syntax throughout this section. We use \(f\) and \(g\) as variables naming functions. Usually, \(i\) and \(j\) will stand for integer heap values, such as \(\text{int}(i)\) or \(\text{rawint}_{\text{pre}}(r)\). In mathematical expressions, \(i\) and \(j\) are given their obvious interpretations as integers.

### 5.1 Evaluation contexts

We specify the evaluation order for the atoms in an expression using evaluation contexts. There are four kinds of contexts, corresponding to atoms \(A\), special-calls \(S\), allocation operations \(L\), and expressions \(E\). Each context is specified as a term with a single “hole” \(\mathbb{[]}\), in which evaluation is to be performed. In general, evaluation is leftmost-innermost. The evaluation contexts are listed in Figure 5.1.

### 5.2 Atom evaluation

The non-value atoms are variables, unary and binary operators, and the polymorphic type operators.

Unary and binary operators are specified using an interpretation \([\text{op}]\) that defines an actual function for each of the operators. We omit the interpretations in this paper, but they are the obvious ones. For example, the interpretation \([\text{+}_{\mathbb{Z}_{31}}]\) for the \(\mathbb{Z}_{31}\) addition operator is actual 31 bit, signed addition. Note that in \texttt{RED-ATOM-UNOP}, we use informal syntax; \(i\) and \(j\) are taken to be atoms with appropriate values for the operator being evaluated. Similar informal syntax is used in \texttt{RED-ATOM-BINOP}.

\[
\begin{align*}
(\Gamma \mid \mathcal{C} \mid E[i]) & \rightarrow (\Gamma \mid \mathcal{C} \mid E[[\text{unop}](i)]) & \text{RED-ATOM-UNOP} \\
(\Gamma \mid \mathcal{C} \mid E[i \text{ binop } j]) & \rightarrow (\Gamma \mid \mathcal{C} \mid E[[\text{binop}](i, j)]) & \text{RED-ATOM-BINOP}
\end{align*}
\]

Variables that are defined as heap values \(h\) can be evaluated. Note that the heap value \(h\) may be another variable.

\[
(\Gamma, v: t = h \mid \mathcal{C} \mid E[v]) \rightarrow (\Gamma, v: t = h \mid \mathcal{C} \mid E[h]) & \text{RED-ATOM-VAR}
\]

Type application is used to instantiate polymorphic functions. The type parameters of the function \(f = \lambda \alpha_1, \ldots, \alpha_m \cdot b\) are defined in a new function \(g\) as the actual type arguments \(u_1, \ldots, u_m\). Note we are assuming the alpha renaming convention to rename type variables as necessary.

\[
((\Gamma', f: u' = \lambda \alpha_1, \ldots, \alpha_m \cdot b) \text{ as } \Gamma) \mid \mathcal{C} \mid E[\text{ty apply}[u](f, u_1, \ldots, u_m)) \rightarrow (\Gamma, \left[s : \omega = u \right]_m^g, g: u = b \mid \mathcal{C} \mid E[g]) & \text{RED-ATOM-APPLY}
\]

The pack operation pairs a value with type arguments to form a value in an existential type \(u = \exists \alpha_1, \ldots, \alpha_m \cdot t\). The main effect of the rule is to add the value in the context.

\[1\text{It would be straightforward to list out a reduction rule for each unary and binary operator.}\]
5.2. ATOM EVALUATION

\[
A ::= \begin{cases} 
[\ ] \\
\text{unop } A \\
A \ h \ \text{binop } a \\
\text{h binop } A 
\end{cases}
\]

\[
S ::= \begin{cases} 
\text{migrate } [j, A, a] a_{\text{fun}} (a_1, \ldots, a_n) \\
\text{migrate } [j, h_p, A] a_{\text{fun}} (a_1, \ldots, a_n) \\
\text{migrate } [j, h_p, h_o] A (a_1, \ldots, a_n) \\
\text{migrate } [j, h_p, h_o] h_{\text{fun}} (h_1, \ldots, a_i, a_{i+1}, \ldots, a_n) \\
\text{atomic } A (a_{\text{const}}, a_1, \ldots, a_n) \\
\text{atomic } v(A, a_1, \ldots, a_n) \\
\text{atomic } v(h_{\text{const}}, h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \\
\text{rollback } [A, a_{\text{const}}] \\
\text{rollback } [h_{\text{level}}, A] \\
\text{commit } [A] a_{\text{fun}} (a_1, \ldots, a_n) \\
\text{commit } [h_{\text{level}}] A (a_1, \ldots, a_n) \\
\text{commit } [h_{\text{level}}] v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) 
\end{cases}
\]

\[
L ::= \langle h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n \rangle_{\text{tuple-class}} : t \\
\text{union}(tv[h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n], t) : t \\
\text{array}(A, a_{\text{init}}) : t \\
\text{array}(h_{\text{size}}, A) : t \\
\text{malloc}(A) : t
\]

\[
E ::= \begin{cases} 
\text{let } v : t = A \text{ in } e \\
\text{let } v : t = (\text{"s" : } t_s)[h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n] \text{ in } e \\
A (a_1, \ldots, a_n) \\
v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \\
\text{special } S \\
\text{match } A \text{ with } [s_i \mapsto c_i]_1^n \\
\text{let } v : L \text{ in } e \\
\text{let } v : t = A[a_2] \text{ in } e \\
\text{let } v : t = h[A] \text{ in } e \\
A[a_2] : t \leftarrow a_3 : e \\
v[A] : t \leftarrow a_3 : e \\
v[h] : t \leftarrow A : e \\
\text{global} : t \leftarrow A : e
\end{cases}
\]

Figure 5.1: Evaluation contexts

\[
((\Gamma', v_1 : u' = b) \text{ as } \Gamma \mid C \mid E[\text{ty-pack}[u](v_1, u_0, \ldots, u_{m-1})]) \rightarrow \text{RED-ATOM-PACK} \\
(\Gamma, v_2 : u = \text{ty-pack}[u](v_1, u_0, \ldots, u_{m-1}) \mid C \mid E[v_2])
\]

The unpack operation is the destructor for a packed value. The new value is added to the context with the actual types \(u_0, \ldots, u_{m-1}\) as arguments for each of the type parameters \(\alpha_0, \ldots, \alpha_{m-1}\).

\[
(\Gamma', v_1 : u = \text{ty-pack}[3[\alpha_0]_0^{m-1}.t](v, [u_i]_0^{m-1})) \text{ as } \Gamma \mid C \mid E[\text{ty-unpack}(v_1)] \rightarrow \text{RED-ATOM-UNPACK} \\
(\Gamma, [\alpha_i : \omega = u_i]_0^{m-1}, v_2 : t = v \mid C \mid E[v_2])
\]

39
5.3 Basic expressions

Expression evaluation is defined on expressions in which the outermost atoms are heap values.

Evaluation of $\text{let } v: u = h \text{ in } e$ adds the definition $v: u = h$ to the context, then evaluates the expression $e$.

$\begin{align*}
(\Gamma | C | \text{let } v: u = h \text{ in } e) & \rightarrow (\Gamma, v: u = h | C | e) \quad \text{RED-LET-ATOM}
\end{align*}$

For external calls, we defer to a semantic interpretation function $["s"]$ that specifies the interpretation of the built-in function named "s".

$\begin{align*}
(\Gamma | C | \text{let } v: t = ("s": (u_1, \ldots, u_n) \rightarrow t)(h_1, \ldots, h_n) \text{ in } e) & \rightarrow (\Gamma, v: t = ["s"](h_1, \ldots, h_n) | C | e) \quad \text{RED-LET-EXT}
\end{align*}$

A tail-call to function $(\lambda v_1, \ldots, v_n. e)$ with arguments $(h_1, \ldots, h_n)$ adds the definitions $[v_i: t_i = h_i]_1^n$ to the evaluation context, and evaluates the expression $e$.

$\begin{align*}
((\Gamma', v: (u_1, \ldots, u_n) \rightarrow t = \lambda v_1, \ldots, v_n. e) & \text{ as } \Gamma | C | v(h_1, \ldots, h_n)) \rightarrow (\Gamma, [v_i: u_i = h_i]_1^n | C | e) \quad \text{RED-TAIL-CALL}
\end{align*}$

5.4 Special-calls

The runtime implementation of special-calls is described in Sections 8.2–8.3. When a process migrates, the entire process (including $\Gamma$, $C$, and the continuation function) migrates to a new location, which is just another runtime environment. The migration itself is effectively invisible to the process, although the result of external calls may reflect the new machine environment. Operationally, evaluation of the expression $\text{migrate } [j, a_{\text{ptr}}, a_{\text{off}}] f(a_1, \ldots, a_n)$ ignores the location described in the string $a_{\text{ptr}}[a_{\text{off}}]$, and acts as a tail-call $f(a_1, \ldots, a_n)$.

$\begin{align*}
(\Gamma | C | \text{special migrate } [j, h_{\text{ptr}}, h_{\text{off}}] f(h_1, \ldots, h_n)) & \rightarrow (\Gamma | C | f(h_1, \ldots, h_n)) \quad \text{RED-SYS-MIGRATE}
\end{align*}$

The runtime implementation of atomic transactions is described in Section 8.3. Transactions are entered with the $\text{atomic } f(a_{\text{const}}, a_1, \ldots, a_n)$ special-call. The runtime adds a process checkpoint to the checkpoint environment $C$ and evaluation proceeds with a tail-call $f(a_{\text{const}}, a_1, \ldots, a_n)$. Operationally, we specify a checkpoint as a pair of a program state $\Gamma$, and a function $f(\phi, a_1, \ldots, a_n)$ to be called if the transaction is aborted. The function’s first argument, written as $\phi$, is the transaction parameter that may be replaced on rollback. Entry into the transaction adds the process checkpoint to the context, then calls $f(a_{\text{const}}, a_1, \ldots, a_n)$ with the transaction parameter $a_{\text{const}}$.

$\begin{align*}
(\Gamma | C | \text{special atomic } f(h_{\text{const}}, h_1, \ldots, h_n)) & \rightarrow (\Gamma | C | f(\phi, h_1, \ldots, h_n); C | f(h_{\text{const}}, h_1, \ldots, h_n)) \quad \text{RED-ATOMIC}
\end{align*}$

When a transaction with checkpoint $\langle \Gamma, f(\phi, a_1, \ldots, a_n) \rangle$ is rolled back with the $\text{rollback } [i, j]$ special-call to level $i$ with transaction parameter $j$, evaluation proceeds as a tail-call $f(j, a_1, \ldots, a_n)$ using the original process context $\Gamma$ and the truncated checkpoint environment $C_i; \ldots; C_j$. All checkpoints with level higher than $i$ are discarded.

---

2The level that was rolled back is re-entered by this primitive; in effect, the state that is restored is the state captured immediately after the level was entered. This is described in further detail in section 8.3.1.
5.5. ALLOCATION EXPRESSIONS

(\Gamma' \mid C_m; \ldots; C_1 = \langle \Gamma, f(\varphi, h_1, \ldots, h_n) \rangle; \ldots; C_1 \mid \text{special rollback} \ [i, j] \rangle) \rightarrow \text{RED-ATOMIC-ROLLBACK-1}

(\Gamma \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, h_1, \ldots, h_n))

\text{when } i \in \{1 \ldots m\}

In most cases, rollback operates on the most recently entered level. As a short-hand, when \( i = 0 \) the runtime will roll back only the most recent level.

(\Gamma' \mid C_m = \langle \Gamma, f(\varphi, h_1, \ldots, h_n) \rangle; \ldots; C_1 \mid \text{special rollback} \ [0, j] \rangle) \rightarrow \text{RED-ATOMIC-ROLLBACK-2}

(\Gamma \mid C_m; \ldots; C_1 \mid f(h_1, \ldots, h_n))

\text{when } i \in \{0 \ldots m\}

It is an error to specify a checkpoint that does not exist.

(\Gamma \mid C_m; \ldots; C_1 \mid \text{special rollback} \ [i, j] \rangle) \rightarrow \text{error}

\text{RED-ATOMIC-ROLLBACK-ERROR}

\text{when } i \notin \{0 \ldots m\} \lor m = 0

When a transaction is committed with the special-call commit \([i] f(a_1, \ldots, a_n)\), the checkpoint is deleted from the checkpoint context, and evaluation continues with a tail-call to the function \( f(a_1, \ldots, a_n) \).

(\Gamma \mid C_m; \ldots; C_1 \mid \text{special commit} \ [i] \ f(h_1, \ldots, h_n) \rangle) \rightarrow \text{RED-ATOMIC-COMMIT-1}

(\Gamma \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n))

\text{when } i \in \{0 \ldots m\}

In most cases, commit operates on the most recently entered level. As a short-hand, when \( i = 0 \) the runtime will commit the most recent level.

(\Gamma \mid C_m; \ldots; C_1 \mid \text{special commit} \ [0] \ f(h_1, \ldots, h_n) \rangle) \rightarrow \text{RED-ATOMIC-COMMIT-2}

(\Gamma \mid C_m; \ldots; C_1 \mid f(h_1, \ldots, h_n))

\text{when } i \notin \{0 \ldots m\} \lor m = 0

It is an error to specify a checkpoint that does not exist.

(\Gamma \mid C_m; \ldots; C_1 \mid \text{special commit} \ [i] \ f(h_1, \ldots, h_n) \rangle) \rightarrow \text{error}

\text{RED-ATOMIC-COMMIT-ERROR}

5.5 Allocation expressions

There are five allocation rules, corresponding to the five aggregate types: tuples, arrays, unions, rawdata, and frames.

When a tuple is allocated, the tuple value is added to the environment.

(\Gamma \mid C \mid \text{let } v = \langle h_1, \ldots, h_n \rangle \text{ : } u \text{ in } e \rangle \rightarrow (\Gamma, v: u = \langle h_1, \ldots, h_n \rangle \mid C \mid e) \text{ RED-ALLOC-TUPLE}

Array allocation is like tuple allocation, but all the elements of the array are initialized with the value \( h_{in} \).

(\Gamma \mid C \mid \text{let } v = \text{array}(i_{\text{size}}, h_{in}) : t \text{ in } e \rangle \rightarrow (\Gamma, v: t = \langle h^{i_{\text{size}}}_{in}, \ldots, h^{i_{\text{size}}}_{in} \rangle \mid C \mid e) \text{ RED-ALLOC-ARRAY}

\text{when } i_{\text{size}} \geq 0

It is an error to allocate an array of negative dimension.

(\Gamma \mid C \mid \text{let } v = \text{array}(i_{\text{size}}, h_{in}) : t \text{ in } e \rangle \rightarrow \text{error}

\text{RED-ALLOC-ARRAY-ERROR}

\text{when } i_{\text{size}} < 0

41
Union constructor allocation adds the value to the environment.

\[ (\Gamma \mid C \mid \text{let } v = \text{union}(t[v[h_1, \ldots, h_n], j] : t \in e) \rightarrow (\Gamma, v : t = t[v[j(h_1, \ldots, h_n)] \mid C \mid e) \]  

\text{RED-ALLOC-UNION}

We do not give precise semantics of rawdata aggregates. The heap value is represented with the term \( (c) \), which represents some possible value for the aggregate. The runtime provides a specific representation, which we do not state explicitly in the rules. Note that rawdata is not initialized explicitly at the time of allocation.

\[ (\Gamma \mid C \mid \text{let } v = \text{malloc}(i_{\text{size}}) : t \in e) \rightarrow (\Gamma, v : t = \langle c \rangle \mid C \mid e) \]  

\text{RED-ALLOC-MALLOC}

\[ \text{when } i_{\text{size}} \geq 0 \]

It is an error to allocate a rawdata aggregate of negative size.

\[ (\Gamma \mid C \mid \text{let } v = \text{malloc}(i_{\text{size}}) : t \in e) \rightarrow \text{error} \]  

\text{RED-ALLOC-MALLOC-ERROR}

\[ \text{when } i_{\text{size}} < 0 \]

Frame allocation is similar to rawdata allocation. The data area is unsafe and uninitialized.

\[ (\Gamma \mid C \mid \text{let } v = \text{frame}(t[v[t_1, \ldots, t_m]] \in e) \rightarrow \]  

\[ (\Gamma, v : \text{frame}(t[v[t_1, \ldots, t_m]] = \langle c : \text{frame}(t[v[t_1, \ldots, t_m]]) \mid C \mid e) \]  

\text{RED-ALLOC-FRAME}

\section{5.6 Match statements}

Match statements evaluate to the first case where the match succeeds. Operationally, there are two cases: the value to be matched is an integer, or the value is of union type.

The integer cases are represented with a single rule. In this case, the side condition

\[ (i \in s_k) \land \forall j \in \{1, \ldots, k-1\}.(i \notin s_j) \]

forces the choice of the first possible match.

\[ (\Gamma \mid C \mid \text{match } i \text{ with } [s_j \mapsto e_j]_{j=1}^n) \rightarrow (\Gamma \mid C \mid e_k) \]  

\text{RED-MATCH-INT}

\[ \text{when } (i \in s_k) \land \forall j \in \{1, \ldots, k-1\}.(i \notin s_j) \]

Matching on values of union type is similar. The actual value has a specific tag \( j \), and the first match case to match \( j \) is chosen. In keeping with the type rule \text{TY-MATCH-UNION}, the type of the value being matched is adjusted to reflect the more specific tag.

\[ (\Gamma, v : \text{union}(t[v[t_1, \ldots, t_m], s] = t[v[j(a_1, \ldots, a_n)] \mid C \mid \text{match } v \text{ with } [s_i \mapsto e_i]_{i=1}^n) \rightarrow \]  

\[ (\Gamma, v : \text{union}(t[v[t_1, \ldots, t_m], s] = t[v[j(a_1, \ldots, a_n)] \mid C \mid e_k) \]  

\text{RED-MATCH-UNION}

\[ \text{when } (j \in s_k) \land \forall p \in \{1, \ldots, k-1\}.(i \notin s_p) \]

\section{5.7 Subscripting operations}

The subscripting operations correspond to the aggregate values: tuples, arrays, unions, rawdata, and frames.

The projection for a tuple \( \langle h_0, \ldots, h_{n-1} \rangle \) and index \( j \) is the element \( h_j \). There are no error cases for tuples\(^3\).

\(^3\)The typing rules for tuple and union subscripting ensure that the index is always within bounds.
5.7. SUBSCRIPTING OPERATIONS

\[(\Gamma', v_2: u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma \mid C \mid \text{let } v_1: u = v_2[j] \text{ in } e \rightarrow \text{RED-LetSub-Tuple}\]
\[(\Gamma, v_1: u = h_j \mid C \mid e)\]

\[(\Gamma', v: u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma \mid C \mid v[j]: u \leftarrow h; e \rightarrow \text{RED-SetSub-Tuple}\]
\[(\Gamma', v: u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle \mid C \mid e)\]

Arrays are similar to tuples when the array index is in bounds.
\[(\Gamma', v_2: u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma \mid C \mid \text{let } v_1: u = v_2[j] \text{ in } e \rightarrow \text{RED-LetSub-Array}\]
\[(\Gamma, v_1: u = h_j \mid C \mid e)\]
\[\text{when } j \in \{0 \ldots n - 1\}\]

\[(\Gamma', v: u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma \mid C \mid v[j]: u \leftarrow h; e \rightarrow \text{RED-SetSub-Array}\]
\[(\Gamma', v: u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle \mid C \mid e)\]
\[\text{when } j \in \{0 \ldots n - 1\}\]

It is an error for an array subscript to be out-of-bounds.
\[(\Gamma', v_2: t_1 = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma \mid C \mid \text{let } v_1: t_1 = v_2[j] \text{ in } e \rightarrow \text{RED-LetSub-Array>Error}\]
\[\text{error}\]
\[\text{when } j \notin \{0 \ldots n - 1\}\]

\[(\Gamma', v: t_1 = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma \mid C \mid v[j]: t_2 \leftarrow h; e \rightarrow \text{RED-SetSub-Array>Error}\]
\[\text{error}\]
\[\text{when } j \notin \{0 \ldots n - 1\}\]

Union values are like tagged tuples. For a value \(tv_j(h_0, \ldots, h_{n-1})\) and index \(k\), the element being accessed is \(h_k\). There are no error cases for unions.
\[(\Gamma', v_2: u' = tv_j(h_0, \ldots, h_{n-1})) \text{ as } \Gamma \mid C \mid \text{let } v_1: u = v_2[k] \text{ in } e \rightarrow \text{RED-LetSub-Union}\]
\[(\Gamma, v_1: u = h_k \mid C \mid e)\]

\[(\Gamma, v: u' = tv_j(h_0, \ldots, h_{n-1}) \mid C \mid v[k]: u \leftarrow h; e) \rightarrow \text{RED-SetSub-Union}\]
\[(\Gamma, v: u' = tv_j(h_0, \ldots, h_{k-1}, h, h_{k+1}, \ldots, h_{n-1}) \mid C \mid e)\]

For the unsafe aggregates, the subscript operations perform a runtime type check. Rather than specify the operations here, we rely on an operational semantics provided by the runtime, described in Chapter 8. For aggregates of data type, we assume the existence of a runtime function \(\text{runtime}(\Gamma \mid c[j]: t)\) that projects a valid value \(h\) of type \(t\) from data array \(c\), or results in an error. We also assume the existence of a runtime function \(\text{runtime}(\Gamma \mid \langle c \rangle[j]: t \leftarrow h)\) to store a value in a rawdata aggregate, returning a new value \(\langle c'\rangle\) or producing an error.

\[\text{runtime}(\Gamma \mid c[j]: t) = \begin{cases} h & \text{the runtime type check } \Gamma \vdash h : t \text{ succeeds} \\ \text{error} & \text{otherwise} \end{cases}\]

\[\text{runtime}(\Gamma \mid \langle c \rangle[j]: t \leftarrow h) = \begin{cases} \langle c'\rangle & \text{the runtime assignment succeeds} \\ \text{error} & \text{otherwise} \end{cases}\]

Given these runtime functions, a rawdata subscript operation returns the value given by the runtime, or produces an error.
\[(\Gamma', v_2: u' = \langle c \rangle) \text{ as } \Gamma \mid C \mid \text{let } v_1 : u = v_2[j] \text{ in } e \rightarrow \text{RED-LetSub-RawData}\]

\[(\Gamma, v_1 : u = h \mid C \mid e)\]
\[\text{when runtime}(\Gamma \mid \langle c \rangle[j] : u) = h\]

\[(\Gamma', v : u' = \langle c \rangle) \text{ as } \Gamma \mid C \mid \text{let } v_1 : u = v_2[j] \text{ in } e \rightarrow \text{RED-LetSub-RawData-Error}\]

\[\text{when runtime}(\Gamma \mid \langle c \rangle[j] : u) = \text{error}\]

\[(\Gamma', v : u' = \langle c \rangle) \text{ as } \Gamma \mid C \mid v[j] : u \leftarrow h ; e \rightarrow \text{RED-SetSub-RawData}\]

\[\Gamma', v : u' = \langle c' \rangle \mid C \mid e\]
\[\text{when runtime}(\Gamma \mid \langle c \rangle[j] : u) = \langle c' \rangle\]

\[(\Gamma', v : u' = \langle c \rangle) \text{ as } \Gamma \mid C \mid v[j] : u \leftarrow h ; e \rightarrow \text{RED-SetSub-RawData-Error}\]

\[\text{when runtime}(\Gamma \mid \langle c \rangle[j] : u) = \text{error}\]

Frame operations are similar to rawdata operations. The frame data is accessed by the runtime functions \[\text{runtime}(\Gamma \mid \langle C : \text{frame}(tv[t_1, \ldots, t_n]) \rangle[j] : t)\] and \[\text{runtime}(\Gamma \mid \langle C : \text{frame}(tv[t_1, \ldots, t_n]) \rangle[j] : h \leftarrow t)\].

\[(\Gamma', v_2 : u' = \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle) \text{ as } \Gamma \mid C \mid \text{let } v_1 : u = v_2[j] \text{ in } e \rightarrow \text{RED-LetSub-Frame}\]

\[(\Gamma, v_1 : u = h \mid C \mid e)\]
\[\text{when runtime}(\Gamma \mid \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle[j] : u) = h\]

\[(\Gamma', v_2 : u' = \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle) \text{ as } \Gamma \mid C \mid \text{let } v_1 : u = v_2[j] \text{ in } e \rightarrow \text{RED-LetSub-Frame-Error}\]

\[\text{when runtime}(\Gamma \mid \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle[j] : u) = \text{error}\]

The subscript assignment function relies on the runtime function \[\text{runtime}(\Gamma \mid \langle C : \text{frame}(tv[t_1, \ldots, t_n]) \rangle[j] : h \leftarrow t)\].

\[(\Gamma', v : u' = \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle) \text{ as } \Gamma \mid C \mid v[j] : u \leftarrow h ; e \rightarrow \text{RED-SetSub-Frame}\]

\[\Gamma', v : u' = \langle c' : \text{frame}(tv[t_1, \ldots, t_n]) \rangle \mid C \mid e\]
\[\text{when runtime}(\Gamma \mid \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle[j] : u) = \langle c' : \text{frame}(tv[t_1, \ldots, t_n]) \rangle\]

\[(\Gamma', v : u' = \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle) \text{ as } \Gamma \mid C \mid v[j] : u \leftarrow h ; e \rightarrow \text{RED-SetSub-Frame-Error}\]

\[\text{when runtime}(\Gamma \mid \langle c : \text{frame}(tv[t_1, \ldots, t_n]) \rangle[j] : u) = \text{error}\]

5.8 Global variable expressions

Global operations fetch and assign values to global labels.

\[(\Gamma', l : u = h) \text{ as } \Gamma \mid C \mid \text{let } v : u = \text{global} \text{ in } e \rightarrow (\Gamma, v : u = h \mid C \mid e) \rightarrow \text{RED-LetGlobal}\]

\[(\Gamma', l : u = h') \text{ as } \Gamma \mid C \mid \text{global} : l : u \leftarrow h ; e \rightarrow (\Gamma', l : u = h \mid C \mid e) \rightarrow \text{RED-SetGlobal}\]
Chapter 6

Basic FIR properties

In this chapter, we develop a few basic lemmas that are needed to prove the preservation and progress (typesafety) theorems. The brief outline is as follows. To prove preservation, we need to show that each rule in the operational semantics does not change the type of the program, and for this we develop a substitution lemma (Lemma 6.4.8) to show that atom evaluation does not change the type of the program. Another basic property is that each well-typed expression has exactly one type (see Section 6.3).

6.1 Well-formedness properties

Initially, we establish that well-formed environments \( \Gamma \) contain only well-typed definitions. We develop this in four parts.

1. The **Well-formed sub-contexts** Lemma 6.1.1 shows that if a context \( \Gamma \) is well-formed, so are its parts.

2. The **Well-formed context types** Lemma 6.1.2 shows that the types in the variable and global definitions in any well-formed context are well-formed types.

3. The **Well-formed contexts** Lemma 6.1.3 shows that the well-formedness judgment \( \Gamma \vdash \phi \) follows from any judgment \( \Gamma \vdash J \).

4. The **Well-formed context value** Lemma 6.1.4 shows that the values in each definition in a well-formed context are also well-formed.

**Lemma 6.1.1** Well-formed sub-contexts If \( \Gamma, d \vdash \phi \) for some definition or declaration \( d \), and \( \text{dom}(d) \cap FV(\Gamma) = \emptyset \), then \( \Gamma \vdash \phi \).

We prove this by induction on the length of the proof of \( \Gamma, d \vdash \phi \). The proof must end with one of the thinning rules listed in Section 4.1 (not including WF-EMPTY-CONTEXT).

**Declarations** Suppose \( d \) is a declaration \( tv : k, v : t, \) or \( l : t \) and the well-formedness proof ends with one of the following rules.
\[\begin{align*}
\Gamma \vdash \phi & \quad \text{THIN-TYPE} \\
\Gamma, tw: k \vdash \phi & \quad \text{THIN-TYPE}\text{-DEF} \\
\Gamma, v: u \vdash \Omega & \quad \text{THIN-VAR} \\
\Gamma, l: u \vdash \Omega & \quad \text{THIN-GLOBAL}
\end{align*}\]

In each of these cases, \(\Gamma \vdash \phi\) follows directly from the premises of the rule. \(\triangle\)

**Definitions** Suppose \(d\) is a definition \(tw: k = tydef, v: t = b,\) or \(l: t = h,\) and the well-formedness proof ends with one of the following rules.

\[\begin{align*}
\Gamma, tv: k \vdash tydef : k & \quad \Gamma, tw: k \vdash \phi \\
\Gamma, tv: k = tydef \vdash \phi & \quad \text{THIN-TYPE}\text{-DEF} \\
\Gamma, v: u = b \vdash \phi & \quad \text{THIN-VAR}\text{-DEF} \\
\Gamma, l: u = h \vdash \phi & \quad \text{THIN-GLOBAL}\text{-DEF}
\end{align*}\]

The judgment \(\Gamma \vdash \phi\) follows by induction and the premises \(\Gamma, tw: k \vdash \phi, \Gamma, v: t \vdash \phi,\) and \(\Gamma, l: t \vdash \phi.\) \(\triangle\)

Otherwise, it must be the case that the well-formedness proof was applied to some other declaration or definition \(d' \in \Gamma.\) Let \(\Gamma'\) be the environment \(\Gamma\) without the definition \(d'.\) That is, \(\Gamma', d = \Gamma.\)

**Declarations** If \(d\) is a declaration \(tv: k, v: t,\) or \(l: t,\) then the well-formedness proof ends with one of the following rules.

\[\begin{align*}
\Gamma', d \vdash \phi & \quad \text{THIN-TYPE} \\
\Gamma', d, tw: k \vdash \phi & \quad \text{THIN-TYPE} \\
\Gamma', d, v: u \vdash \Omega & \quad \text{THIN-VAR} \\
\Gamma', d, l: u \vdash \Omega & \quad \text{THIN-GLOBAL}
\end{align*}\]

The judgment \(\Gamma' \vdash \phi\) follows from the induction hypothesis, and the premises \(\Gamma', d \vdash \phi,\) and we can infer that \(\Gamma', d' \vdash \phi\) by reapplying the well-formedness rule. \(\triangle\)

**Definitions** If \(d'\) is a definition \(tw: k = t, v: t = h,\) or \(l: t = h,\) then the well-formedness proof ends with one of the following rules.

\[\begin{align*}
\Gamma', d, tv: k + tydef : k & \quad \Gamma', d, tv: k \vdash \phi \\
\Gamma', d, tv: k = tydef \vdash \phi & \quad \text{THIN-TYPE}\text{-DEF} \\
\Gamma', d, v: u + b : u & \quad \Gamma', d, v: u \vdash \phi \\
\Gamma', d, v: u = b \vdash \phi & \quad \text{THIN-VAR}\text{-DEF} \\
\Gamma', d, l: u + h : u & \quad \Gamma', d, l: u \vdash \phi \\
\Gamma', d, l: u = h \vdash \phi & \quad \text{THIN-GLOBAL}\text{-DEF}
\end{align*}\]
Let $d'$ be the declaration $tv: k, v: t, or l: t$ that corresponds to the definition $d'$. We can infer that \( \Gamma', d' \vdash \emptyset \) by induction and the second premise of each rule. In each case, the definition’s value is well-typed by the first premise, and we can reapply the rule to conclude that $\Gamma', d \vdash \emptyset$.

This concludes the proof of the Well-formed sub-contexts Lemma 6.1.1.

\[ \square \]

**Lemma 6.1.2** Well-formed context types If $\Gamma \vdash \emptyset$ then for each declaration $v: t, l: t$, or each definition $v: t = b, l: t = b$ in $\Gamma$, the term $t$ is a well-formed type.

We prove this by induction on the length of the proof of $\Gamma \vdash \emptyset$.

**Empty context** If $\Gamma$ is empty, it contains no declarations or definitions, and the result follows trivially.

**Type definitions** Suppose the proof ends with a type definition rule.

\[
\begin{align*}
\Gamma & \vdash \emptyset \\
\Gamma, tv: k & \vdash \emptyset \\
\Gamma, tv: k & \vdash tydef: k \\
\Gamma, tv: k = tydef & \vdash \emptyset \\
\end{align*}
\]

By induction, we can infer that each variable and global definition in $\Gamma$ is defined over a well-formed type. Since the declaration $tv: k$ and definition $tv: k = t$ is not a variable and global definition, the result follows directly.

**Variable and global declarations** Suppose the proof ends with one of the following rules.

\[
\begin{align*}
\Gamma & \vdash \emptyset \\
\Gamma, u & \vdash \Omega \\
\Gamma, \tau: u & \vdash \emptyset \\
\end{align*}
\]

Using the induction hypothesis on the premise $\Gamma \vdash \emptyset$ we can conclude that each variable and global definition in $\Gamma$ is defined over a well-formed type. Since the second premise requires that the term $u$ be a well-formed type, the result follows directly.

**Variable and global definitions** Suppose the proof ends with one of the following rules.

\[
\begin{align*}
\Gamma, v: u & \vdash b: u \\
\Gamma, v: u & \vdash \emptyset \\
\Gamma, l: u & \vdash b: u \\
\Gamma, l: u & \vdash \emptyset \\
\end{align*}
\]

In both of these cases, we can use the induction hypothesis on the second premise to conclude that the type $u$ is well-formed.

\[ \triangle \]
This concludes the proof of well-formed context types. \qed

For the next lemma, we show that for any judgment $\Gamma \vdash J$, the context $\Gamma$ must be well-formed.

**Lemma 6.1.3** **Well-formed contexts** A proof of the judgment $\Gamma \vdash \top$ follows from any judgment $\Gamma \vdash J$.

We prove this by the length of the proof $\Gamma \vdash J$. We consider the proofs of $\Gamma \vdash J$ where $J \neq \top$, and the proof of $\Gamma \vdash J$ does not end in the use of a rule where one of the premises has the form $\Gamma \vdash J'$ for some $J'$ (otherwise the result follows trivially). The rules that fit this description are listed below.

\[
\begin{align*}
\Gamma, tv; k & \vdash tydef : k \quad \Gamma, tv; k \vdash J & \text{Thin-Type-Def} \\
\Gamma, tv; k & \vdash k = tydef \vdash J & \\
\Gamma, v; u + b : u & \Gamma, v; u \vdash J & \text{Thin-Var-Def} \\
\Gamma, l; u + h : u & \Gamma, l; u \vdash J & \text{Thin-Global-Def} \\
\Gamma, [\alpha_i : \omega]_1^n \vdash \text{tydef}_s : k_s & \\
\Gamma \vdash \lambda x_1, \ldots, x_m. \text{tydef}_s : \omega^m \to k_s & \\
\Gamma, \{ u_i : t_i \}_i^n \vdash e : t & \text{Ty-StoreFun-Mono} \\
\Gamma \vdash \lambda x_1, \ldots, x_m. \lambda x_1, \ldots, x_n. e : (u_1, \ldots, u_n) \to t & \\
\Gamma \vdash \lambda x_1, \ldots, x_m. \lambda x_1, \ldots, x_n. e : \forall x_1, \ldots, x_m. (u_1, \ldots, u_n) \to t & \text{Ty-StoreFun-Poly} \\
\Gamma, [\alpha_i : \omega]_1^n \vdash t : \Omega & \text{WF-ST-TyExists} \\
\Gamma \vdash \exists \alpha_1, \ldots, \alpha_m. \lfloor t_1, \ldots, t_n \rfloor \to u & \\
\Gamma \vdash \forall \alpha_1, \ldots, \alpha_m. \lfloor t_1, \ldots, t_n \rfloor \to u : \omega & \text{WF-ST-TyFun-Poly} \\
\{ s_i \in \text{set}_i \}_i^n & \neq \text{union} \left( a \in [\text{set}_i]_i^n \right) & \left[ \Gamma, v : \text{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t \right]_i^n & \text{Ty-Match-Union} \\
\end{align*}
\]

**Subcontexts** If the proof ends with the rule $\text{WF-TyDef}$, $\text{Ty-StoreFun-Poly}$, $\text{WF-ST-TyExists}$, or $\text{WF-ST-TyFun-Poly}$, we can conclude by induction that $\Gamma, [\alpha_i : \omega]_1^n \vdash \top$.

Similarly, if the proof ends with the rule $\text{Ty-StoreFun-Mono}$, we can conclude by induction that $\Gamma, [v_i : t_i]_i^n \vdash \top$.

The result $\Gamma \vdash \top$ follows directly by application of the **Well-formed sub-contexts** Lemma 6.1.1. \triangle

**Definitions** From the three cases $\text{Thin-Type-Def}$, $\text{Thin-Var-Def}$, and $\text{Thin-Global-Def}$, let the definitions and declarations be defined as in the following table.

\[
\begin{array}{ccc}
\text{decl} & \text{def} & \text{Rule} \\
\text{tv: } k & \text{tv: } k = t & \text{Thin-Type-Def} \\
v : t & v : t = b & \text{Thin-Var-Def} \\
l : t & l : t = h & \text{Thin-Global-Def} \\
\end{array}
\]

48
6.1. WELL-FORMEDNESS PROPERTIES

We can conclude \( \Gamma, \text{decl} \vdash \phi \) by induction, using the second premise of each rule. Since the first premise requires well-formedness of the definition’s value, we can reapply the rule to conclude that \( \Gamma, \text{def} \vdash \phi \).

\( \triangle \)

**Unions** For **Ty-Match-Union**, we can conclude from the premise \([\Gamma, v : \text{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t_i]_1^n\), that the context \( \Gamma, v : \text{union}(tv[t_1, \ldots, t_m], s_i) \) is well-formed for all of the match cases \([s_i \mapsto e_i]_1^n\). The side condition requires that there be at least one match case, and we can use the induction hypothesis on the first case to conclude that \( \Gamma, v : \text{union}(tv[t_1, \ldots, t_m], s_1) \vdash \phi \).

From the **Well-formed Context Types** Lemma 6.1.2, the type \( \text{union}(tv[t_1, \ldots, t_m], s_1) \) must be well-formed. That is, there is a proof \( \Gamma \vdash \text{union}(tv[t_1, \ldots, t_m], s_1) : k_i \). The only rule that can be used directly to prove well-formedness is the **WF-ST-TyUnion** rule.

\[
\begin{array}{c}
\text{set}_1 \subseteq \{0 \ldots n - 1\} \quad \Gamma \vdash [t_i : \omega]_1^n \\
\Gamma \vdash tv[t_1, \ldots, t_m] : \omega_\text{union}[
\text{set}_1] \\
\hline
\Gamma \vdash \text{union}(tv[t_1, \ldots, t_m], \text{set}_1) : \omega
\end{array}
\]

The only condition on \( \text{set}_1 \) is \( \text{set}_1 \subseteq \{0 \ldots n - 1\} \), and we can infer the type \( \text{union}(tv[t_1, \ldots, t_m], s_\text{union}) \) is also well-formed. We can infer from the **Well-formed Sub-contexts** Lemma 6.1.1 that \( \Gamma \vdash \phi \), and we can conclude that \( \Gamma, v : \text{union}(tv[t_1, \ldots, t_m], s_\text{union}) \vdash \phi \) from the **Thin-Var** rule.

\[
\begin{array}{c}
\Gamma \vdash \phi \\
\Gamma, v : u : \Omega \\
\hline
\Gamma, v : u : \phi
\end{array}
\]

\( \triangle \)

This concludes the proof of the **Well-formed Contexts** Lemma 6.1.3.

\( \Box \)

**Lemma 6.1.4** **Well-formed Context Values** If \( \Gamma \vdash \phi \), then every value in each definition in \( \Gamma \) is well-typed. That is, all of the following hold.

- For each type definition \( tv : k = t \) in \( \Gamma \), \( \Gamma \vdash t : k \).
- For each variable definition \( v : t = b \) in \( \Gamma \), \( \Gamma \vdash b : t \).
- For each global definition \( l : t = h \) in \( \Gamma \), \( \Gamma \vdash h : t \).

We prove this by induction on the length of the proof of \( \Gamma \vdash \phi \).

**Empty Context** The empty context contains no definitions, so the result follows trivially. \( \triangle \)

**Declarations** Suppose the well-formedness proof ends with one of the declaration forms.

\[
\begin{array}{c}
\Gamma \vdash \phi \\
\hline
\Gamma, tv : k \vdash \phi \quad \text{Thin-Type}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \phi \\
\Gamma, v : u : \Omega \\
\hline
\Gamma, v : u : \phi \quad \text{Thin-Var}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \phi \\
\Gamma, l : u : \phi \\
\hline
\Gamma, l : u : \phi \quad \text{Thin-Global}
\end{array}
\]

We can conclude that each definition in \( \Gamma \) is well-formed from the premise \( \Gamma \vdash \phi \). \( \triangle \)
Definitions Suppose the well-formedness proof ends with one of the definition forms.

\[
\begin{align*}
\Gamma, tv : k \vdash tydef : k & \quad \Gamma, tv : k \vdash \varphi : \top \\
\Gamma, tv : k & = tydef \vdash \varphi : \top & \text{THIN-TYPE-DEF} \\
\Gamma, v : u \vdash b : u & \quad \Gamma, v : u \vdash \varphi : \top & \text{THIN-VAR-DEF} \\
\Gamma, l : u \vdash h : u & \quad \Gamma, l : u \vdash \varphi : \top & \text{THIN-GLOBAL-DEF}
\end{align*}
\]

In each case, the first premise of the rule requires that the definition's value be well-formed. △

This concludes the proof of well-formed context values. □

6.2 Proof induction

Before continuing, we first need a technical lemma that we use frequently throughout our reasoning. We often use induction on the length of a proof \( \Gamma \vdash \varphi : t \), where \( \varphi \) is an atom \( a \), expression \( e \), store value \( b \), or global label \( l \). These proofs require that we enumerate the cases, relying on the proof structure to establish the use of particular rules. For example, if we know that there is a proof of \( \Gamma \vdash \text{let} \ v : u = a \ \text{in} \ e : t \), we would argue that the final rule in the proof must be the \textsc{Ty-LetAtom} rule, because no other rule applies.

\[
\begin{align*}
\Gamma & \vdash a : u \\
\Gamma, v : u & \vdash e : t \\
\Gamma & \vdash \text{let} \ v : u = a \ \text{in} \ e : t
\end{align*}
\]

Unfortunately, this is not exactly true. There are seven other rules that can terminate the proof, shown in Figure 6.1.

\[
\begin{align*}
\Gamma & \vdash \varphi : t_1 \\
\Gamma, tv : k & \vdash \varphi : t_1 & \text{THIN-TYPE} \\
\Gamma & \vdash \varphi : t_1 \quad \Gamma & \vdash t_2 : \Omega \\
\Gamma, v : t_2 & \vdash \varphi : t_1 & \text{THIN-VAR} \\
\Gamma & \vdash \varphi : t_1 \quad \Gamma & \vdash t_2 : \Omega \\
\Gamma, l : t_2 & \vdash \varphi : t_1 & \text{THIN-GLOBAL} \\
\Gamma, tv : k & \vdash tydef : k \quad \Gamma, tv : k & \vdash \varphi : t_1 \\
\Gamma, v : t_2 & \vdash \varphi : t_1 & \text{THIN-TYPE-DEF} \\
\Gamma, v : t_2 & \vdash b : t_2 & \Gamma, v : t_2 & \vdash \varphi : t_1 \\
\Gamma, v : t_2 & \vdash b = \varphi : t_1 & \text{THIN-VAR-DEF} \\
\Gamma, l : t_2 & \vdash h : t_2 & \Gamma, l : t_2 & \vdash \varphi : t_1 \\
\Gamma, l : t_2 & \vdash h = \varphi : t_1 & \text{THIN-GLOBAL-DEF} \\
\Gamma & \vdash t_1 = t_2 : k & \Gamma & \vdash \varphi : t_2 \\
\Gamma & \vdash \varphi : t_1 & \text{TYP-SUBST}
\end{align*}
\]

Figure 6.1: Thinning and substitution rules

In the following lemma, we show that the use of these rules does not affect the proof in any substantial way.
Lemma 6.2.1 Proof Induction Assume \( \Gamma \vdash \varphi : t \), where \( \varphi \) is an atom \( a \), expression \( e \), store value \( b \), or global label \( l \), and let \( R \) be the set of rules in Figure 6.1. Then the proof of \( \Gamma \vdash \varphi : t \) contains the application of a rule \( r \not\in R \) with conclusion \( \hat{\Gamma} \vdash \varphi : t' \) for some context \( \hat{\Gamma} \) where \( \hat{\Gamma} \vdash t = t' : \Omega \). Furthermore, for each premise of \( r \) of the form \( \hat{\Gamma}, \Delta \vdash J \), there is a derivation \( \Gamma, \Delta \vdash J \).

We prove this by induction on the length of the proof of \( \Gamma \vdash \varphi : t \).

Base case Suppose the final rule in the proof is the use of a rule \( r \not\in R \). The result follows trivially. \( \square \)

Declarations Suppose the final rule in is the application of one of the rules \textsc{Thin-Type}, \textsc{Thin-Var}, or \textsc{Thin-Global}. These rules are similar; we illustrate the argument with the \textsc{Thin-Var} rule.

\[
\Gamma \vdash \varphi : t \quad \Gamma \vdash u : \Omega \\
\overline{\quad \Gamma, v : u \vdash \varphi : t}
\]

We can conclude from the induction hypothesis and the premise \( \Gamma \vdash \varphi : t \) that the proof contains the use of a rule \( r \not\in R \) for the abbreviated context \( \hat{\Gamma} = \Gamma \). For any derivation \( \hat{\Gamma}, \Delta \vdash J \), the context is well-formed \( \hat{\Gamma}, \Delta \vdash \varphi \) by the \textsc{Well-Formed Contexts} Lemma 6.1.3, and we can infer that \( \hat{\Gamma}, \Delta \vdash u : \Omega \) from the premise \( \hat{\Gamma}, u : \Omega \vdash \varphi \) and application of the thinning rules. The derivation \( \Gamma, v : u, \Delta \vdash J \) follows by re-applying the \textsc{Thin-Var} rule. \( \square \)

Definitions Suppose the final rule in the proof is the application of one of the rules \textsc{Thin-Type-Def}, \textsc{Thin-Var-Def}, or \textsc{Thin-Global-Def}. These rules are similar; we illustrate the argument with the \textsc{Thin-Var-Def} rule.

\[
\Gamma, v : u \vdash b : u \\
\Gamma, v : u \vdash \varphi : t \\
\overline{\quad \Gamma, v : u = b \vdash \varphi : t}
\]

We can conclude from the induction hypothesis and the premise \( \Gamma, v : t \vdash \varphi : t \) that the proof contains the use of a rule \( r \not\in R \) for the abbreviated context \( \hat{\Gamma} = \Gamma, v : t \). For any derivation \( \hat{\Gamma}, \Delta \vdash J \), the context is well-formed \( \hat{\Gamma}, \Delta \vdash \varphi \) by the \textsc{Well-Formed Contexts} Lemma 6.1.3, and we can infer that \( \hat{\Gamma}, \Delta \vdash b : t \), from the premise \( \hat{\Gamma} \vdash b : t \) and repeated application of the thinning rules. Thus, for any derivation \( \hat{\Gamma}, \Delta \vdash J \), we can re-apply the \textsc{Thin-Var-Def} rule to conclude that \( \Gamma, v : t = b \vdash J \). \( \square \)

Substitution Suppose the final rule in the proof is an application of \textsc{Ty-Subst}.

\[
\Gamma \vdash t = u : k \\
\Gamma \vdash \varphi : u \\
\overline{\quad \Gamma \vdash \varphi : t}
\]

Since the types \( t \) and \( u \) are equal, we can conclude that the derivation of \( \Gamma \vdash \varphi : u \) contains the use of a rule \( r \not\in R \), and the context \( \Gamma \) is unchanged. \( \square \)

This concludes the \textsc{Proof Induction} Lemma 6.2.1. \( \square \)
6.3 Type uniqueness

In this section, we show that each value and each expression in a program has a unique type.

Lemma 6.3.1 Uniqueness of atom types If $\Gamma \vdash a : t_1$ and $\Gamma \vdash a : t_2$ then $t_1 = t_2$.

To prove this lemma, we construct a deterministic function $\text{typeof}(\Gamma, a)$ that produces the type of the atom $a$, given the context $\Gamma$ and the atom.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$\text{typeof}(\Gamma, a)$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{enum}_i(j)$</td>
<td>$\text{enum}_i$</td>
<td>$\text{TY-ATOM\text{ENUM}}$</td>
</tr>
<tr>
<td>$\text{int}(i)$</td>
<td>$\mathbb{Z}_{\text{sign}}$</td>
<td>$\text{TY-ATOM\text{INT}}$</td>
</tr>
<tr>
<td>$\text{rawint}_{\text{pre}}(r)$</td>
<td>$\mathbb{Z}_{\text{sign}}$</td>
<td>$\text{TY-ATOM\text{RAWINT}}$</td>
</tr>
<tr>
<td>$\text{float}_{\text{pre}}(x)$</td>
<td>$\mathbb{R}_{\text{sign}}$</td>
<td>$\text{TY-ATOM\text{FLOAT}}$</td>
</tr>
<tr>
<td>$\text{label}(tv, rt, r_{\text{off}})$</td>
<td>$\mathbb{Z}_{\text{2}}$</td>
<td>$\text{TY-ATOM\text{LABEL}}$</td>
</tr>
<tr>
<td>$\text{sizeof}(tv_1, \ldots, tv_n, r_{\text{size}})$</td>
<td>$\mathbb{Z}_{\text{2}}$</td>
<td>$\text{TY-ATOM\text{SIZEOF}}$</td>
</tr>
<tr>
<td>$v$</td>
<td>$\Gamma(v)$</td>
<td>$\text{TY-ATOMVAR}$</td>
</tr>
<tr>
<td>$\text{const}[t](tv, i)$</td>
<td>$t$</td>
<td>$\text{TY-ATOM\text{CONST}}$</td>
</tr>
<tr>
<td>$\text{ty_apply}[t](a, t_1, \ldots, t_n)$</td>
<td>$t$</td>
<td>$\text{TY-ATOM\text{TY_APPLY}}$</td>
</tr>
<tr>
<td>$\text{ty_pack}[t](v, t_1, \ldots, t_n)$</td>
<td>$t$</td>
<td>$\text{TY-ATOM\text{TY_PACK}}$</td>
</tr>
<tr>
<td>$\text{ty_unpack}(v)$</td>
<td>$(\Gamma(v))[v, i/\alpha_i]_{\text{2}}^{n-1}$</td>
<td>$\text{TY-ATOM\text{TY_UNPACK}}$</td>
</tr>
<tr>
<td>$\text{unop}\ a$</td>
<td>$\text{res}(\text{unop})$</td>
<td>$\text{TY-ATOM\text{UNOP}}$</td>
</tr>
<tr>
<td>$a_1 \text{ binop} a_2$</td>
<td>$\text{res}(\text{binop})$</td>
<td>$\text{TY-ATOM\text{BINOP}}$</td>
</tr>
</tbody>
</table>

The rules listed are the only typing rules for atoms. A straightforward inspection of the rules shows that if $\Gamma \vdash a : t$, then $t = \text{typeof}(\Gamma, a)$.

Lemma 6.3.2 Uniqueness of expression types If $\Gamma \vdash e_r : t$ then $t = \text{enum}_0$.

We prove this by induction on the length of the proof of $\Gamma \vdash e_r : t$. We use the Proof Induction Lemma 6.2.1 throughout.

Let forms Suppose $e_r$ is any of the let forms, including $\text{TY-LET\text{ATOM}}$, $\text{TY-LET\text{EXT}}$, $\text{TY-LET\text{SUB-TUPLE}}$, $\text{TY-LET\text{SUB-ARRAY}}$, $\text{TY-LET\text{SUB-RAWDATA}}$, $\text{TY-LET\text{SUB-UNION}}$, $\text{TY-LET\text{SUB-FRAME}}$, $\text{TY-ALLOC}$, and $\text{TY-LET\text{GLOBAL}}$. These all have similar proofs. We illustrate with the proof for $\text{TY-LET\text{ATOM}}$.

Suppose $e_r = \text{let} v : u = a \text{ in } e$. The proof of typing for $e_r$ must use the rule $\text{TY-LET\text{ATOM}}$.

\[
\begin{align*}
\overrightarrow{\Gamma} & \vdash a : t \\
\overrightarrow{\Gamma}, v : u \vdash e : t \\
\overrightarrow{\Gamma} & \vdash \text{let} v : u = a \text{ in } e : t
\end{align*}
\]

From the premise $\overrightarrow{\Gamma}, v : u \vdash e : t$, we can conclude that $t = \text{enum}_0$ by induction.

TailCall Suppose $e_r = a_f(a_1, \ldots, a_n)$. The proof of typing for $e_r$ must use the rule $\text{TY-TAIL\text{CALL}}$. 
6.3. TYPE UNIQUENESS

\[ \widehat{\Gamma} \vdash a_j : (u_1, \ldots, u_n) \rightarrow \text{enum}_0 \quad \widehat{\Gamma} \vdash [a_i : u_i]_1^n \]

\[ \widehat{\Gamma} \vdash a_j(a_1, \ldots, a_n) : \text{enum}_0 \]

The only possible type for a tail-call is \(\text{enum}_0\), so \(t = \text{enum}_0\). \(\triangle\)

**Special-calls** Suppose \(e_r = \text{special tailop}\). The proof of typing for \(e_r\) must use the Ty-Special-Call rule.

\[ \widehat{\Gamma} \vdash \text{tailop} : \text{special} t \]

\[ \widehat{\Gamma} \vdash \text{special tailop} : t \]

There are four rules that can prove the premise \(\widehat{\Gamma} \vdash \text{tailop} : \text{special} t\): Ty-SysMigrate, Ty-Atomic, Ty-AtomicCommit, and Ty-AtomicRollback. In each of these cases \(t = \text{enum}_0\). \(\triangle\)

**Match** The match rules Ty-Match-Int, Ty-Match-Enum, Ty-Match-RawInt, Ty-Match-Union all have similar forms. We illustrate the proof for these cases with Ty-Match-Union.

In this case, the proof of \(\widehat{\Gamma} \vdash e_r : t\) has the following form.

\[ \left[ s_i \in \text{set}_l \right]_1^n \setminus \{ \} \neq s_{\text{union}} = \bigcup_{i=1}^{n} s_i \quad \widehat{\Gamma}, v : \text{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t \]

\[ \widehat{\Gamma}, v : \text{union}(tv[t_1, \ldots, t_m], s_{\text{union}}) \vdash \text{match} \; v \; \text{with} \; [s_i \rightarrow e_i]_1^n : t \]

The side-condition \(\emptyset \neq s_{\text{union}}\) requires there be at least one premise \(\widehat{\Gamma}, v : \text{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t\), and we can conclude that \(t = \text{enum}_0\) by induction. \(\triangle\)

**Subscript Assignment** The rules for assignment operations Ty-SetSub-Tuple, Ty-SetSub-RawData, Ty-SetSub-Union, Ty-SetSub-Array, and Ty-SetSub-Frame have similar forms. We illustrate with Ty-SetSub-Tuple.

\[ \widehat{\Gamma} \vdash a_1 : (u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1})_e \quad \widehat{\Gamma} \vdash a_3 : u \quad \widehat{\Gamma} \vdash e : t \]

\[ \widehat{\Gamma} \vdash a_1[\text{offset}_c(\text{const}(j))]: u \leftarrow a_3; \; e : t \]

The premise \(\widehat{\Gamma} \vdash e : t\) requires that \(t = \text{enum}_0\) by induction. \(\triangle\)

**Assignment** The type judgment for an assignment operation Ty-SetGlobal has the following form.

\[ \widehat{\Gamma} \vdash l : u \quad \widehat{\Gamma} \vdash a : u \quad \widehat{\Gamma} \vdash e : t \]

\[ \widehat{\Gamma} \vdash \text{global}l : u \leftarrow a; \; e : t \]

The premise \(\widehat{\Gamma} \vdash e : t\) requires that \(t = \text{enum}_0\) by induction. \(\triangle\)

This concludes the proof of type uniqueness for expressions. \(\square\)
6.4 Substitution

Substitution is the primary lemma needed for proving the type preservation theorem 7.1.1. The goal in this section is to show that if an expression \( E[a_1] : t \) is well-typed for some value \( a_1 : u \), then \( E[a_2] : t \) for any other atom \( a_2 : u \).

We prove this in two parts. First, we show that if \( E[a] \) is well-typed, then \( a : t_a \) for some type \( t_a \). Second, we use this to prove the substitution lemma.

**Lemma 6.4.1** Atom Typing If \( \Gamma_r \vdash A[a] : t_r \), then there is a type \( t_a \) such that \( \Gamma_r \vdash a : t_a \).

We prove this by structural induction on \( A \).

**Trivial context** Suppose \( A = [] \). Then \( A[a] = a \) and we have the following judgment.

\[
\Gamma_r \vdash a : t_r.
\]

\( \triangle \)

**Unary operators** Suppose \( A[a] = \text{unop} A'[a] \). The type judgment must use the Ty-ATOMUNOP rule.

\[
\frac{
\widehat{\Gamma_r} \vdash A'[a] : \text{arg}(\text{unop}) \quad \widehat{\Gamma_r} \vdash \text{res}(\text{unop}) : \Omega
}{
\widehat{\Gamma_r} \vdash \text{unop} A'[a] : \text{res}(\text{unop})
}
\]

Using induction on the premise \( \Gamma_r \vdash A'[a] : \text{arg}(\text{unop}) \) we infer that there is some type \( t_a \) for which

\[
\Gamma_r \vdash a : t_a.
\]

\( \triangle \)

**Binary operators** Suppose \( A[a] = A'[a] \text{ binop } a' \). The type judgment must use the Ty-ATOMBINOP rule.

\[
\frac{
\widehat{\Gamma_r} \vdash A'[a] : \text{arg}_1(\text{binop}) \quad \widehat{\Gamma_r} \vdash a' : \text{arg}_2(\text{binop}) \quad \widehat{\Gamma_r} \vdash \text{res}(\text{binop}) : \Omega
}{
\widehat{\Gamma_r} \vdash A'[a] \text{ binop } a' : \text{res}(\text{binop})
}
\]

Using induction on the premise \( \Gamma_r \vdash A'[a] : \text{arg}_1(\text{binop}) \), we infer that there is some type \( t_a \) for which

\[
\Gamma_r \vdash a : t_a.
\]

The case \( A[a] = h \text{ binop } A'[a] \) is similar.

\( \triangle \)

This concludes the Atom Typing Lemma 6.4.1.

\( \Box \)

**Lemma 6.4.2** Special-Call Typing If \( \Gamma_r \vdash S[a] : t_r \), then there is a type \( t_a \) such that \( \Gamma_r \vdash a : t_a \).

We prove this by case analysis on \( S \).
Migrate Suppose $S[a]$ is one of the following forms.

$$S ::= \text{migrate} [i, A, a_0] a_{fun}(a_1, \ldots, a_n)$$
$$| \text{migrate} [i, h_p, A] a_{fun}(a_1, \ldots, a_n)$$
$$| \text{migrate} [i, h_p, h_0] A(a_1, \ldots, a_n)$$
$$| \text{migrate} [i, h_p, h_0] h_{fun}(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n)$$

The type judgment for $S$ must use the rule $\text{TY-MIGRATE}$.

$$\Gamma_r \vdash a_{ptr} : \text{data}$$
$$\Gamma_r \vdash a_{off} : \mathbb{Z}_{32}^{\text{signed}}$$

$$\Gamma_r \vdash a_i : t_i^n$$
$$\Gamma_r \vdash a_f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0$$

$\Gamma_r \vdash \text{migrate} [j, a_{ptr}, a_{off}] a_f(a_1, \ldots, a_n) : \text{special enum}_0$

We have the following cases for the type judgment $\Gamma_r \vdash A[a] : t_A$.

- If $A = a_{ptr}$, then $t_A = \text{data}$.
- If $A = a_{off}$, then $t_A = \mathbb{Z}_{32}^{\text{signed}}$.
- If $A = a_f$, then $t_A = (t_1, \ldots, t_n) \rightarrow \text{enum}_0$.
- If $A = a_i$, then $t_A = t_i$.

From the $\text{ATOM TYPING}$ Lemma 6.4.1, we can infer that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$. △

Atomic Suppose $S[a]$ is one of the following forms.

$$S ::= \text{atomic} A(a_{const}, a_1, \ldots, a_n)$$
$$| \text{atomic} v(A, a_1, \ldots, a_n)$$
$$| \text{atomic} v(h_{const}, h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n)$$

The type judgment for $S$ must use the rule $\text{TY-ATOMIC}$.

$$\Gamma_r \vdash a_{const} : \mathbb{Z}_{32}^{\text{signed}}$$
$$\Gamma_r \vdash [a_i : t_i]_1^n$$
$$\Gamma_r \vdash a_f : (\mathbb{Z}_{32}^{\text{signed}}, t_1, \ldots, t_n) \rightarrow \text{enum}_0$$

$\Gamma_r \vdash \text{atomic} a_f(a_{const}, a_1, \ldots, a_n) : \text{special enum}_0$

We have the following case for the type judgment $\Gamma_r \vdash A[a] : t_A$.

- If $A = a_f$, then $t_A = (\mathbb{Z}_{32}^{\text{signed}}, t_1, \ldots, t_n) \rightarrow \text{enum}_0$.
- If $A = a_{const}$, then $t_A = \mathbb{Z}_{32}^{\text{signed}}$.
- If $A = a_i$, then $t_A = t_i$.

From the $\text{ATOM TYPING}$ Lemma 6.4.1, we can infer that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$. △

Atomic Rollback Suppose $S[a]$ is one of the following forms.

$$S ::= \text{rollback} [A, a_{const}]$$
$$| \text{rollback} [h_{level}, A]$$

The type judgment for $S$ must use the rule $\text{TY-ATOMIC-ROLLBACK}$.

55
\[
\begin{align*}
\Gamma_r &\vdash a_{\text{level}} : \mathbb{Z}^{\text{signed}}_{32} & \Gamma_r &\vdash a_{\text{const}} : \mathbb{Z}^{\text{signed}}_{32} \\
& \hline \\
\Gamma_r &\vdash \text{rollback} \ [a_{\text{level}}, a_{\text{const}}] : \text{special enum}_0
\end{align*}
\]

In both cases \(A = a_{\text{const}}\) and \(A = a_{\text{level}}\), we have the judgment \(\Gamma_r \vdash A[a] : \mathbb{Z}^{\text{signed}}_{32}\). From the \textsc{Atom Typing} Lemma 6.4.1, we can infer that there is some type \(t_a\) for which \(\Gamma_r \vdash a : t_a\). \(\triangle\)

**Atomic Commit** Suppose \(S[a]\) is one of the following forms.

\[
S ::= \text{commit} \ [A] \ a_{\text{fun}}(a_1, \ldots, a_n) \\
| \text{commit} \ [h_{\text{level}}] \ A(a_1, \ldots, a_n) \\
| \text{commit} \ [h_{\text{level}}] \ v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n)
\]

The type judgment for \(S\) must use the rule \textsc{Ty-AtomicCommit}.

\[
\begin{align*}
\Gamma_r &\vdash a_{\text{level}} : \mathbb{Z}^{\text{signed}}_{32} & \Gamma_r &\vdash [a_i : t_i]_1^n & \Gamma_r &\vdash a_f : (t_1, \ldots, t_n) \to \text{enum}_0 \\
\hline \\
\Gamma_r &\vdash \text{commit} \ [a_{\text{level}}] \ a_f(a_1, \ldots, a_n) : \text{special enum}_0
\end{align*}
\]

We have the following cases for the type judgment \(\Gamma_r \vdash A[a] : t_A\).

- If \(A = a_{\text{fun}}\), then \(t_A = (t_1, \ldots, t_n) \to \text{enum}_0\).
- If \(A = a_{\text{level}}\), then \(t_A = \mathbb{Z}^{\text{signed}}_{32}\).
- If \(A = a_i\), then \(t_A = t_i\).

From the \textsc{Atom Typing} Lemma 6.4.1, we can infer that there is some type \(t_a\) for which \(\Gamma_r \vdash a : t_a\). \(\triangle\)

This concludes the \textsc{Special-Call Typing} Lemma 6.4.2.

\(\square\)

**Lemma 6.4.3 Allocation Typing** If \(\Gamma_r \vdash L[a] : t_r\), then there is a type \(t_a\) such that \(\Gamma_r \vdash a : t_a\).

We prove this by case analysis on \(L\).

**Alloc tuple** Suppose \(L = \langle h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n \rangle_{\text{tuple\_class}} : t\). The type judgment must use the rule \textsc{Ty-Alloc-Tuple}.

\[
\begin{align*}
\Gamma_r &\vdash [a_i : u_i]_1^n & \Gamma_r &\vdash \langle u_1, \ldots, u_n \rangle_c : \omega \\
\hline \\
\Gamma_r &\vdash \langle a_1, \ldots, a_n \rangle_c : \text{alloc} \langle u_1, \ldots, u_n \rangle_c
\end{align*}
\]

From the premise \(\Gamma_r \vdash a_i : t_i\), we can infer that \(\Gamma_r \vdash A[a] : t_i\). From the \textsc{Atom Typing} Lemma 6.4.1, we can infer that there is some type \(t_a\) for which \(\Gamma_r \vdash a : t_a\). \(\triangle\)

**Alloc union** Suppose \(L = \text{union}(tv[h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n], i) : t\). The type judgment must use the rule \textsc{Ty-Alloc-Union}.

\[
\begin{align*}
\Gamma_r &\vdash tv(a_1, \ldots, a_k) : \text{union}(tv[t_1, \ldots, t_m], \{j\}) \\
\hline \\
\Gamma_r &\vdash \text{union}(tv[a_1, \ldots, a_k], j) : \text{alloc union}(tv[t_1, \ldots, t_m], \{j\})
\end{align*}
\]

\(56\)
The derivation of the premise must use the store-typing rule Ty-STORE-UNION for unions.

\[
\begin{align*}
\Gamma_r, [\alpha_i : \omega = t_i^m] & \vdash [\alpha_i : u_i^k]_1 \\
\Gamma_r \vdash \text{union}(tw[t_1', \ldots, t_m'], \{j\}) : \Omega \\
\Gamma_r \vdash tw[t_1', \ldots, t_m'] = t_0 + \cdots + t_{j-1} + \langle u_1, \ldots, u_k \rangle + t_{j+1} + \cdots + t_{n-1} : \omega_{\text{union}[n]} \\
\Gamma_r \vdash tw(a_1, \ldots, a_k) : \text{union}(tw[t_1', \ldots, t_m'], \{j\})
\end{align*}
\]

From the premise \( \Gamma_r \vdash a_i : t_i \), we can infer that \( \Gamma_r \vdash A[a_i] : t_i \). From the ATOM TYPING Lemma 6.4.1, we can infer that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a \).

**Alloc array** Suppose \( L \) is one of the following.

\[
L :: \quad \text{array}(A, a_{\text{init}}) : t \\
\mid \quad \text{array}(h_{\text{size}}, A) : t
\]

The type judgment must use the rule Ty-ALLOC-ARRAY.

\[
\begin{align*}
\Gamma_r \vdash a_{\text{size}} : \text{offset} \\
\Gamma_r \vdash a_{\text{init}} : u
\end{align*}
\]

\[
\Gamma_r \vdash \text{array}(a_{\text{size}}, a_{\text{init}}) : \text{alloc} u \text{ array}
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A \).

- If \( A = a_{\text{size}} \), then \( t_A = \text{offset} \).
- If \( A = a_{\text{init}} \), then \( t_A = u \).

From the ATOM TYPING Lemma 6.4.1, we can infer that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a \).

**Malloc** Suppose \( L = \text{malloc}(A) : t \). The type judgment must use the rule Ty-ALLOC-MALLOC.

\[
\Gamma_r \vdash A[a] : \text{offset}
\]

\[
\Gamma_r \vdash \text{malloc}(A[a]) : \text{alloc data}
\]

From the premise \( \Gamma_r \vdash A[a] : \text{offset} \), and the ATOM TYPING Lemma 6.4.1, we can infer that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a \).

This concludes the ALLOCATION TYPING Lemma 6.4.3.

**Lemma 6.4.4** **Expression Typing** If \( \Gamma_r \vdash E[a] : t_r \), then there is a type \( t_a \) such that \( \Gamma_r \vdash a : t_a \).

We prove this by a case analysis on \( E \).

**LetAtom** Suppose \( E = \text{let } v : t = A \text{ in } e \). The type judgment must use the Ty-LETATOM rule.

\[
\begin{align*}
\Gamma_r \vdash A[a] : u \\
\Gamma_r, v : u \vdash e : t_r
\end{align*}
\]

\[
\Gamma_r \vdash \text{let } v : u = A[a] \text{ in } e : t_r
\]

From the premise \( \Gamma_r \vdash A[a] : t_1 \), and the ATOM TYPING Lemma 6.4.1, we can infer that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a \).
**FIR Safety**

**CHAPTER 6. BASIC FIR PROPERTIES**

**LetExt** Suppose $E = \text{let } v \triangleq t = \left( \text{"s" } \mapsto t_s \right) \left( h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_{n-1} \right) \text{ in } e$. The type judgment must use the TY-LetExt rule.

\[
\begin{array}{l}
\Gamma_r \vdash \left[ a_i : u_i \right]^n_i \\
\Gamma_r, v : u \vdash e : t_r \\
\Gamma_r \vdash \left[ \text{"s" } \mapsto u \right] : (u_1, \ldots, u_n) \rightarrow u \\
\Gamma_r \vdash \text{let } v : u = \left( \text{"s" } \mapsto (u_1, \ldots, u_n) \rightarrow u \right) (a_1, \ldots, a_n) \text{ in } e : t_r.
\end{array}
\]

From the premise $\Gamma_r \vdash a_i : u_i$, we can infer that $\Gamma_r \vdash A[a] : u_i$. From the **ATOM TYPING** Lemma 6.4.1, we can infer that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$. \(\triangle\)

**TailCall** Suppose $E$ is one of the following.

\[
E ::= A(a_1, \ldots, a_n) \\
\quad \mid v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_{n-1})
\]

The type judgment must use the TY-TailCall rule.

\[
\begin{array}{l}
\Gamma_r \vdash a_j : (u_1, \ldots, u_n) \rightarrow \text{enum}_0 \\
\Gamma_r \vdash [a_i : u_i]^n_i \\
\Gamma_r \vdash a_j(a_1, \ldots, a_n) : \text{enum}_0 \\
\end{array}
\]

We have the following cases for the type judgment $\Gamma_r \vdash A[a] : t_A$.

- If $A = a_j$, then $t_A = (u_1, \ldots, u_n) \rightarrow \text{enum}_0$.
- If $A = a_i$, then $t_A = u_i$.

From the **ATOM TYPING** Lemma 6.4.1, we can infer that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$. \(\triangle\)

**Special Call** Suppose $E = \text{special } S$. The type judgment must use the TY-SpecialCall rule.

\[
\begin{array}{l}
\Gamma_r \vdash S : \text{special } t_r \\
\Gamma_r \vdash \text{special } S : t_r
\end{array}
\]

From the premise $S : \text{special } t$ and the **SPECIAL-CALL TYPING** Lemma 6.4.2, we can infer that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$. \(\triangle\)

**Match** Suppose $E = \text{match } A \text{ with } \left[ s_i \mapsto e_i \right]^n_i$. The type judgment must use the one of the match rules.

- If the rule is TY-Match-INT

\[
\left[ s_i \in \text{set } j \right]^n_i \bigcup_{i=1}^n s_i = \mathbb{Z}_{\geq 1} \\
\Gamma_r \vdash A : \mathbb{Z}_{\geq 1} \\
\Gamma_r \vdash \left[ e_i : t_r \right]^n_i \\
\Gamma_r \vdash \text{match } A \text{ with } \left[ s_i \mapsto e_i \right]^n_i : t_r
\]

then we can infer from the premise $A : \mathbb{Z}_{\geq 1}$ and the **ATOM TYPING** Lemma 6.4.1 that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$.

- If the rule is TY-Match-ENUM

\[
\begin{array}{l}
\left[ s_i \in \text{set } j \right]^n_i \\
\bigcup_{i=1}^n s_i = \text{enum}_j \\
\Gamma_r \vdash A : \text{enum}_j \\
\Gamma_r \vdash \left[ e_i : t_r \right]^n_i \\
\Gamma_r \vdash \text{match } A \text{ with } \left[ s_i \mapsto e_i \right]^n_i : t_r
\end{array}
\]

then we can infer from the premise $A : \text{enum}_j$ and the **ATOM TYPING** Lemma 6.4.1 that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$.

58
• If the rule is Ty-Match-RawInt

\[
[s_i \in \text{set}_1]^n \bigcup_{i=1}^{n} s_i = \mathbb{Z}\_\text{sign}^{\text{pre}} \quad \Gamma_{r} \vdash A : \mathbb{Z}\_\text{sign}^{\text{pre}} \quad \Gamma_{r} \vdash [e_i : t_r]_1^n
\]

then we can infer from the premise \(A : \mathbb{Z}\_\text{sign}^{\text{pre}}\) and the Atom Typing Lemma 6.4.1 that there is some type \(t_a\) for which \(\Gamma_{r} \vdash a : t_a\).

• The typing rule Ty-Match-Union

\[
\begin{align*}
\Gamma_{r}, v : \text{union}(tv[t_1, \ldots, t_m], s_{\text{union}}) & \vdash \text{match} v \text{ with } [s_i \mapsto e_i]_1^n : t_r \\
\end{align*}
\]

requires that \(A = v\). We can infer from the declaration \(v : \text{union}(tv[u_1, \ldots, u_m], s_{\text{union}})\) that \(A : \text{union}(tv[u_1, \ldots, u_m], s_{\text{union}})\). From the Atom Typing Lemma 6.4.1 that there is some type \(t_a\) for which \(\Gamma_{r} \vdash a : t_a\).

\[\triangle\]

Allocation Suppose \(E = \text{let} v = L \text{ in } e\). The type judgment must use the Ty-Alloc rule.

\[
\begin{align*}
\Gamma_{r} \vdash L : \text{alloc} u \quad \Gamma_{r}, v : u \vdash e : t_r \\
\Gamma_{r} \vdash \text{let} v = L \text{ in } e : t_r
\end{align*}
\]

From the premise \(L : \text{alloc} u\) and the Allocation Typing Lemma 6.4.3, we can infer that there is some type \(t_a\) for which \(\Gamma_{r} \vdash a : t_a\).

\[\triangle\]

Subscripting Suppose \(E\) is one of the following forms.

\[
E ::= \text{let} v : t = A[a_2] \text{ in } e \\
| \quad \text{let} v : t = h[A] \text{ in } e
\]

The type judgment must use one of the subscript rules.

• Suppose the rule is Ty-LetSub-RawData.

\[
\begin{align*}
\Gamma_{r} \vdash a_1 : \text{data} \quad \Gamma_{r} \vdash a_2 : \text{offset} \\
\Gamma_{r}, v : u \vdash e : t_r \\
\Gamma_{r} \vdash \text{let} v : u = a_1[a_2] \text{ in } e : t_r
\end{align*}
\]

We can infer from the premises \(a_1 : \text{data}, a_2 : \text{offset}\), and the Atom Typing Lemma 6.4.1 that there is some type \(t_a\) for which \(\Gamma_{r} \vdash a : t_a\).

• Suppose the rule is Ty-LetSub-Array.

\[
\begin{align*}
\Gamma_{r} \vdash a_1 : u \text{ array} \\
\Gamma_{r} \vdash a_2 : \text{offset} \\
\Gamma_{r}, v : u \vdash e : t_r \\
\Gamma_{r} \vdash \text{let} v : u = a_1[a_2] \text{ in } e : t_r
\end{align*}
\]

We can infer from the premises \(a_1 : t_1 \text{ array}, a_2 : \text{offset}\), and the Atom Typing Lemma 6.4.1 that there is some type \(t_a\) for which \(\Gamma_{r} \vdash a : t_a\).
• Suppose the rule is **TY-LETSub-TUPLE**.

\[
\Gamma_r \vdash a_1 : (u_0, \ldots, u_{j-1}, u_j, u_{j+1}, \ldots, u_{n-1}) \quad \Gamma_r, v_1 : u \vdash e : t_r
\]

\[
\Gamma_r \vdash \text{let } v_1 : u = a_1[\text{offset}^\text{const}_r(j)] \text{ in } e : t_r
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A \).

- If \( A = a \), then \( t_A = \langle u_0, \ldots, u_{j-1}, t_1, u_{j+1}, \ldots, u_{n-1} \rangle_c \).
- If \( A = \text{offset}^\text{const}_r(i) \), then \( t_A = \mathbb{Z}_{31} \).

In both cases, we can infer from the **ATOM TYPING** Lemma 6.4.1 that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a \).

• Suppose the rule is **TY-LETSub-UNION**.

\[
\Gamma_r \vdash a_1 : \text{union}(tv[t'_1, \ldots, t'_m], \{j\})
\]

\[
\Gamma_r \vdash tv[t'_1, \ldots, t'_m] = \left( \left[ \bar{u}_j \right]_0^{-1} + \left[ \bar{u}'_j \right]_0^{k-1}, u, \left[ \bar{u}'_j \right]_{k+1}^{l-1} + \left[ \bar{u}_j \right]_{j+1}^{n-1} \right) : \omega_{\text{union}[n]}
\]

\[
\Gamma_r, v : u \vdash e : t_r
\]

\[
\Gamma_r \vdash \text{let } v : u = a_1[\text{offset}^\text{const}_r(k)] \text{ in } e : t_r
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A \).

- If \( A = a_1 \), then \( t_A = \text{union}(tv[t'_1, \ldots, t'_m], \{j\}) \).
- If \( A = \text{offset}^\text{const}_r(i) \), then \( t_A = \mathbb{Z}_{31} \).

In both cases, we can infer from the **ATOM TYPING** Lemma 6.4.1 that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a \).

• Suppose the rule is **TY-LETSub-FRAME**.

\[
\Gamma_r \vdash a_1 : \text{frame}(tv[t'_1, \ldots, t'_m])
\]

\[
\Gamma_r \vdash tv[t'_1, \ldots, t'_m] = \{ t' : \{ t'' : \{ \ldots \} \} \} : \omega_{\text{frame}}
\]

\[
\Gamma_r, v : u \vdash e : t_r
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A \).

- If \( A = a_1 \), then \( t_A = \text{frame}(tv[t'_1, \ldots, t'_m]) \). In this case, we can infer from the **ATOM TYPING** Lemma 6.4.1 that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a \).
- The case \( A = \text{label}(tv, t'1, t'l', \text{r_off}) \), is not allowed, because it is not a valid context.

\[\triangle\]

**Assignment** Suppose \( E \) is one of the following forms.

\[
E \ ::= \ A[a_2] : t \leftarrow a_3 ; e \\
| \ v[A] : t \leftarrow a_3 ; e \\
| \ v[h] : t \leftarrow A ; e
\]

The type judgment must use one of the subscript assignment rules.
• Suppose the rule is \( \text{TY-SETSUB-RAWDATA}. \)

\[
\Gamma_r \vdash a_1 : \text{data} \quad \Gamma_r \vdash a_2 : \text{offset} \quad \Gamma_r \vdash a_3 : u \quad \Gamma_r \vdash e : t_r
\]

\[
\Gamma_r \vdash a_1[a_2] : u \leftarrow a_3; e : t_r
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A. \)

- If \( A = a_1 \), then \( t_A = \text{data}. \)
- If \( A = a_2 \), then \( t_A = \text{offset}. \)
- If \( A = a_3 \), then \( t_A = t_1. \)

We can infer from the \textsc{Atom Typing} Lemma 6.4.1 that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a. \)

• Suppose the rule is \( \text{TY-SETSUB-ARRAY}. \)

\[
\Gamma_r \vdash a_1 : u \text{ array} \quad \Gamma_r \vdash a_2 : \text{offset} \quad \Gamma_r \vdash a_3 : u \quad \Gamma_r \vdash e : t_r
\]

\[
\Gamma_r \vdash a_1[a_2] : u \leftarrow a_3; e : t_r
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A. \)

- If \( A = a_1 \), then \( t_A = u \text{ array}. \)
- If \( A = a_2 \), then \( t_A = \text{offset}. \)
- If \( A = a_3 \), then \( t_A = u. \)

We can infer from the \textsc{Atom Typing} Lemma 6.4.1 that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a. \)

• Suppose the rule is \( \text{TY-SETSUB-TUPLE}. \)

\[
\Gamma_r \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \Gamma_r \vdash a_3 : u \quad \Gamma_r \vdash e : t_r
\]

\[
\Gamma_r \vdash a_1[\text{offset}_c^{\text{const}}(j)] : u \leftarrow a_3; e : t_r
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A. \)

- If \( A = a_1 \), then \( t_A = \langle u_0, \ldots, u_{j-1}, t_1, u_{j+1}, \ldots, u_{n-1} \rangle_c. \)
- If \( A = a_2 \), then \( t_A = t_1. \)
- The case \( A = \text{offset}_c^{\text{const}}(i) \) is not allowed because it is not a valid context.

For the first two cases, we can infer from the \textsc{Atom Typing} Lemma 6.4.1 that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a. \)

• Suppose the rule is \( \text{TY-SETSUB-UNION}. \)

\[
\Gamma_r \vdash a_1 : \text{union}(tv[t'_1, \ldots, t'_m], \{j\})
\]

\[
\Gamma_r \vdash tv[t'_1, \ldots, t'_m] = \left( \left[ u_0 \right]_0^{j-1} + \left[ u'_0 \right]_0^{k-1} + \left[ u'_1 \right]_{k+1}^{l-1} + \left[ u'_i \right]_{j+1}^{n-1} \right) : \omega_{\text{union}[n]}
\]

\[
\Gamma_r \vdash a_3 : u
\]

\[
\Gamma_r \vdash e : t_r
\]

\[
\Gamma_r \vdash a_1[\text{offset}_n^{\text{const}}(k)] : u \leftarrow a_3; e : t_r
\]

We have the following cases for the type judgment \( \Gamma_r \vdash A[a] : t_A. \)

- If \( A = a_1 \), then \( t_A = \text{union}(tv[t'_1, \ldots, t'_m], \{j\}). \)
- If \( A = a_3 \), then \( t_A = t_1. \)
- The case \( A = \text{offset}_n^{\text{const}}(i) \) is not allowed because it is not a valid context.

For the first two cases, we can infer from the \textsc{Atom Typing} Lemma 6.4.1 that there is some type \( t_a \) for which \( \Gamma_r \vdash a : t_a. \)
• Suppose the rule is Ty-SetSub-Frame.

\[
\begin{align*}
& r_{off} \in \mathbb{Z}_{\text{signed}} \\
& \Gamma_r \vdash a_1 : \text{frame}(tv[t_1, \ldots, t_m]) \\
& \Gamma_r \vdash tv[t_1, \ldots, t_m] = \{t' : \{t' : \cdots\} : \omega}_{\text{frame}} \\
& \Gamma_r \vdash a_3 : u \\
& \Gamma_r \vdash e : t_r \\
\end{align*}
\]

\[
\Gamma_r \vdash a_1[\text{label}(tv, t', i_{\text{off}})]: u \leftarrow a_3; e : t_r
\]

We have the following cases for the type judgment $\Gamma_r \vdash A[a] : t_A$.

- If $A = a_1$, then $t_A = \text{frame}(tv[t'_1, \ldots, t'_m])$.
- If $A = a_3$, then $t_A = t_1$.
- The case $A = \text{label}(tv, t', i_{\text{off}})$ is not allowed because it is not a valid context.

For the first two cases, we can infer from the ATOM TYPING Lemma 6.4.1 that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$.

\[\triangle\]

**Global Assignment** Suppose $E = \text{global} : t \leftarrow A; e$. The proof for type judgment must invoke the rule Ty-SetGlobal.

\[
\begin{align*}
\Gamma_r & \vdash l : u \\
\Gamma_r & \vdash A : u \\
\Gamma_r & \vdash e : t_r \\
\Gamma_r & \vdash \text{global} : u \leftarrow A; e : t_r
\end{align*}
\]

From the premise $A : u$, we can infer from the ATOM TYPING Lemma 6.4.1 that there is some type $t_a$ for which $\Gamma_r \vdash a : t_a$.

\[\triangle\]

The concludes the EXPRESSION TYPING Lemma 6.4.4.

\[\square\]

The following four lemmas spell out the different substitution lemmas needed for preservation. In each case, the argument is by structural induction on an evaluation context $\bullet[a_r]$, using the typing rules to show that if $\bullet[a_r]$ is well-typed, then so is $a_r$.

**Lemma 6.4.5 ATOM Substitution** Assume the following:

• $\Gamma \vdash A[a_r] : t$,

• $\Gamma \vdash a_r : u$,

• $\Gamma \vdash a_c : u$.

Then $\Gamma \vdash A[a_c] : t$.

We prove this by structural induction on $\Gamma \vdash A[a_r]$.

**Simple context** Suppose $A = [\]$. Then $A[a_r] = a_r$ and $A[a_c] = a_c$. By the UNIQUENESS OF ATOM TYPES Lemma 6.3.1, $t = u$.

\[\triangle\]
**Unary operations** Suppose \( A = \text{unop} A' \). The proof \( \Gamma \vdash A[a_r]: t \) must use the rule \( \text{Ty-AtomUnop} \), where \( t = \text{res(unop)} \).

\[
\frac{\hat{\Gamma} \vdash A'[a_r]: \text{arg(unop)} \quad \hat{\Gamma} \vdash \text{res(unop)}: \Omega}{\hat{\Gamma} \vdash \text{unop} A'[a_r]: \text{res(unop)}}
\]

By induction, we know \( \Gamma \vdash A'[a_r]: \text{arg(unop)} \), from which we can infer that \( \Gamma \vdash \text{unop} A'[a_r]: \text{res(unop)} \). \( \triangle \)

**Binary operations** Suppose \( A = A' \text{ binop} a \). The proof \( \Gamma \vdash A[a_r]: t \) must use the rule \( \text{Ty-AtomBinop} \).

\[
\frac{\hat{\Gamma} \vdash A'[a_r]: \text{arg}_1(\text{binop}) \quad \hat{\Gamma} \vdash a: \text{arg}_2(\text{binop}) \quad \hat{\Gamma} \vdash \text{res(binop)}: \Omega}{\hat{\Gamma} \vdash A'[a_r] \text{ binop} a : \text{res(binop)}}
\]

By induction, we know \( \Gamma \vdash A'[a_r]: \text{arg}_1(\text{binop}) \), from which we can infer that \( \Gamma \vdash A[a_r]: \text{res(binop)} \). The argument for the context \( A = h \text{ binop} A'' \) is similar. \( \triangle \)

This concludes the proof of the \( \text{Atom Substitution} \) Lemma 6.4.5. \( \square \)

**Lemma 6.4.6 SpecialCall Substitution** Assume all of the following:

- \( \Gamma \vdash S[a_r]: \text{special} t \)
- \( \Gamma \vdash a_r: u \)
- \( \Gamma \vdash a_r: u \)

Then \( \Gamma \vdash S[a_r]: \text{special} t \).

We prove this by structural induction on \( S[a_r] \).

**Migrate** Suppose \( S \) is one of the following.

\[
S ::=	ext{ migrate } [i, A, a_o] a_{f_{\text{in}}}(a_1, \ldots, a_n) \\
| \text{ migrate } [i, h_p, A] a_{f_{\text{in}}}(a_1, \ldots, a_n) \\
| \text{ migrate } [i, h_p, h_o] A(a_1, \ldots, a_1) \\
| \text{ migrate } [i, h_p, h_o] h_{f_{\text{in}}}(h_1, \ldots, h_i, A, a_{i+1}, \ldots, a_n)
\]

The type judgment must use the rule \( \text{Ty-SysMigrate} \).

\[
\begin{align*}
\hat{\Gamma} & \vdash a_{\text{ptr}}: \text{data} \\
\hat{\Gamma} & \vdash a_{\text{off}}: \text{signed} \\
\hat{\Gamma} & \vdash [a_i : t_i]_1^n \\
\hat{\Gamma} & \vdash a_f: (t_1, \ldots, t_n) \rightarrow \text{enum}_0 \\
\overline{\hat{\Gamma}} & \vdash \text{migrate } [j, a_{\text{ptr}}, a_{\text{off}}] a_f(a_1, \ldots, a_n) : \text{special} \text{enum}_0
\end{align*}
\]

We have the following cases for the type judgment \( \Gamma \vdash A[a_r]: t_A \).
• If $A = a_{\text{ptr}}$, then $t_A = \text{data}$.
• If $A = a_{\text{off}}$, then $t_A = \mathbb{Z}^{\text{signed}}_{32}$.
• If $A = a_{\text{fun}}$, then $t_A = (t_1, \ldots, t_n) \rightarrow \text{enum}_0$.
• If $A = a_i$, then $t_A = t_i$.

From the Atom Substitution Lemma 6.4.5, we can conclude that $\Gamma \vdash A[a_i] : t_A$, from which we can infer that $\Gamma \vdash S[a_i] : \text{special}$.

Atomic Suppose $S$ is one of the following.

$$S ::= \text{atomic } A(a_{\text{const}}, a_1, \ldots, a_n)$$
$$\quad | \quad \text{atomic } v(A, a_1, \ldots, a_n)$$
$$\quad | \quad \text{atomic } v(h_{\text{const}}, h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n)$$

The type judgment must use the rule Ty-Atomic.

$$\Gamma \vdash a_{\text{const}} : \mathbb{Z}^{\text{signed}}_{32} \quad \Gamma \vdash [a_i : t_i]^n \quad \Gamma \vdash a_f : (\mathbb{Z}^{\text{signed}}_{32}, t_1, \ldots, t_n) \rightarrow \text{enum}_0$$

$$\Gamma \vdash \text{atomic } a_f(a_{\text{const}}, a_1, \ldots, a_n) : \text{special enum}_0$$

We have the following case for the type judgment $\Gamma \vdash A[a] : t_A$.

• If $A = a_f$, then $t_A = (\mathbb{Z}^{\text{signed}}_{32}, t_1, \ldots, t_n) \rightarrow \text{enum}_0$.
• If $A = a_{\text{const}}$, then $t_A = \mathbb{Z}^{\text{signed}}_{32}$.
• If $A = a_i$, then $t_A = t_i$.

From the Atom Substitution Lemma 6.4.5, we can conclude that $\Gamma \vdash A[a_i] : t_A$, from which we can infer that $\Gamma \vdash S[a_i] : \text{special}$.

Atomic Rollback Suppose $S$ is one of the following.

$$S ::= \text{rollback } [A, a_{\text{const}}]$$
$$\quad | \quad \text{rollback } [h_{\text{level}}, A]$$

The type judgment for $S$ must use the rule Ty-ATOMIC Rollback.

$$\Gamma \vdash a_{\text{level}} : \mathbb{Z}^{\text{signed}}_{32} \quad \Gamma \vdash a_{\text{const}} : \mathbb{Z}^{\text{signed}}_{32}$$

$$\Gamma \vdash \text{rollback } [a_{\text{level}}, a_{\text{const}}] : \text{special enum}_0$$

In both cases: $A[a_r] = a_{\text{const}}$ and $A[a_v] = a_{\text{level}}$, we have the judgment $\Gamma \vdash A[a_r] : \mathbb{Z}^{\text{signed}}_{32}$.

From the Atom Substitution Lemma 6.4.5, we can conclude that $\Gamma \vdash A[a_r] : t_A$, from which we can infer that $\Gamma \vdash S[a_r] : \text{special}$.

Atomic Commit Suppose $S$ is one of the following forms.

$$S ::= \text{commit } [A] a_{\text{fun}}(a_1, \ldots, a_n)$$
$$\quad | \quad \text{commit } [h_{\text{level}}] A(a_1, \ldots, a_n)$$
$$\quad | \quad \text{commit } [h_{\text{level}}] v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n)$$
6.4. SUBSTITUTION

The type judgment for $S$ must use the rule Ty-AtomicCommit:

\[
\Gamma \vdash a_{\text{level}} : \mathbb{Z}_{32}^{\text{signed}} \quad \hat{\Gamma} \vdash [a_i : t_i]_1^n \quad \hat{\Gamma} \vdash a_f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0
\]

\[
\hat{\Gamma} \vdash \text{commit} [a_{\text{level}}] a_f(a_1, \ldots, a_n) : \text{special enum}_0
\]

We have the following cases for the type judgment $\Gamma \vdash A[a_r] : t_A$.

- If $A = a_f$, then $t_A = (t_1, \ldots, t_n) \rightarrow \text{enum}_0$.
- If $A = a_{\text{level}}$, then $t_A = \mathbb{Z}_{32}^{\text{signed}}$.
- If $A = a_i$, then $t_A = t_i$.

From the Atom Substitution Lemma 6.4.5, we can conclude that $\Gamma \vdash A[a_r] : t_A$, from which we can infer that $\Gamma \vdash S[a_r] : \text{special}$.

This concludes the proof of the SpecialCall Substitution Lemma 6.4.6. $\square$

Lemma 6.4.7 Allocation Substitution Assume all of the following.

- $\Gamma \vdash L[a_r] : \text{alloc} t$.
- $\Gamma \vdash a_r : u$.
- $\Gamma \vdash a_e : u$.

Then $\Gamma \vdash L[a_r] : \text{alloc} t$.

We prove this by structural induction on $L$.

Alloc tuple Suppose $L = \langle h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n \rangle_{\text{tuple}} : t$. The type judgment must use the rule Ty-Alloc-Tuple.

\[
\hat{\Gamma} \vdash [a_i : u_i]_1^n \quad \hat{\Gamma} \vdash \langle u_1, \ldots, u_n \rangle_c : \omega
\]

\[
\hat{\Gamma} \vdash \langle u_1, \ldots, u_n \rangle_c : \text{alloc} \langle u_1, \ldots, u_n \rangle_c
\]

From the premise $\Gamma \vdash a_i : u_i$, we can infer that $\Gamma \vdash A[a_r] : u_i$. From the Atom Substitution Lemma 6.4.5, we can conclude that $\Gamma \vdash A[a_r] : u_i$, from which we can infer that $\Gamma \vdash L[a_r] : \text{alloc} t$. $\square$

Alloc union Suppose $L = \text{union}(tv[h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n], i) : t$. The type judgment must use the rule Ty-Alloc-Union.

\[
\hat{\Gamma} \vdash tv_j(a_1, \ldots, a_k) : \text{union}(tv[t_1, \ldots, t_m], \{j\})
\]

\[
\hat{\Gamma} \vdash \text{union}(tv[a_1, \ldots, a_k], j) : \text{alloc union}(tv[t_1, \ldots, t_m], \{j\})
\]

The proof of the premise must use the Ty-StoreUnion rule.
\[ \hat{\Gamma}, [\alpha_i : \omega = t'_i]_1^m \vdash [a_i : u_i]_1^k \]
\[ \hat{\Gamma} \vdash \text{union}(tv[t'_1, \ldots, t'_m], \{ j \}) : \Omega \]
\[ \hat{\Gamma} \vdash tv[t'_1, \ldots, t'_m] = t_0 + \cdots + t_{j-1} + \langle u_1, \ldots, u_k \rangle + t_{j+1} + \cdots + t_{n-1} : \omega_{\text{union}[n]} \]
\[ \hat{\Gamma} \vdash tv_j(a_1, \ldots, a_k) : \text{union}(tv[t'_1, \ldots, t'_m], \{ j \}) \]

From the premise \( \Gamma, [\alpha_i : \omega = t'_i]_1^m \vdash a_i : u_i \), we can infer that \( \Gamma, [\alpha_i : \omega = t'_i]_1^m \vdash A[a_i] : u_i \). From the \textsc{Atom Substitution} Lemma 6.4.5, we can conclude that \( \Gamma, [\alpha_i : \omega = t'_i]_1^m \vdash A[a_c] : u_i \), from which we can infer that \( \Gamma \vdash L[a_c] : \text{alloc} \ t \).

**Alloc array** Suppose \( L \) is one of the following.

\[
L ::= \text{array}(A, a_{\text{init}}) : t \\
| \quad \text{array}(h_{\text{size}}, A) : t
\]

The type judgment must use the rule \textsc{Ty-Alloc-Array}.

\[ \hat{\Gamma} \vdash a : \text{offset} \quad \hat{\Gamma} \vdash \text{a}_{\text{init}} : u \]
\[ \hat{\Gamma} \vdash \text{array}(a_{\text{size}}, a_{\text{init}}) : \text{alloc} \ u \ \text{array} \]

We have the following cases for the type judgment \( \Gamma \vdash A[a_c] : t_A \).

- If \( A[a_r] = a_{\text{size}} \), then \( t_A = \text{offset} \).
- If \( A[a_r] = a_{\text{init}} \), then \( t_A = t' \).

From the \textsc{Atom Substitution} Lemma 6.4.5, we can conclude that \( \Gamma \vdash A[a_c] : t_A \), from which we can infer that \( \Gamma \vdash L[a_c] : \text{alloc} \ t \).

**Malloc** Suppose \( L = \text{malloc}(A) : t \). The type judgment must use the rule \textsc{Ty-Alloc-Malloc}.

\[ \hat{\Gamma} \vdash A[a_r] : \text{offset} \]
\[ \hat{\Gamma} \vdash \text{malloc}(A[a_r]) : \text{alloc} \ data \]

From the premise \( \Gamma \vdash A[a_r] : \text{offset} \), and the \textsc{Atom Substitution} Lemma 6.4.5, we can conclude that \( \Gamma \vdash A[a_c] : \text{offset} \), from which we can infer that \( \Gamma \vdash L[a_c] : \text{alloc} \ t \).

This concludes the proof of \textsc{Allocation Substitution} Lemma 6.4.7.

**Lemma 6.4.8 Substitution** Assume all of the following.

- \( \Gamma \vdash E[a_r] : t \).
- \( \Gamma \vdash a_r : u \).
- \( \Gamma \vdash a_c : u \).

Then \( \Gamma \vdash E[a_c] : t \).
We prove this by using a case analysis on the context $E$.

**LetAtom** Suppose $E = \textbf{let} \ v \colon t_1 = A \ \textbf{in} \ e$. The typing proof for $E$ must use the rule Ty-LetAtom.

$$
\Gamma \vdash A[a_r] : u \\
\Gamma, v : u \vdash e : t \\
\Gamma \vdash \textbf{let} \ v : u = A[a_c] \ \textbf{in} \ e : t
$$

From the Atom Substitution Lemma 6.4.5, and the premise $\Gamma \vdash A[a_r] : t_1$, we know $\Gamma \vdash A[a_c] : t_1$, and we can infer that $\Gamma \vdash \textbf{let} \ v : t_1 = A[a_c] \ \textbf{in} \ e : t$. $\triangle$

**LetExt** Suppose $E = \textbf{let} \ v_1 : t_1 = ("s^n" : t_2) \ (h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \ \textbf{in} \ e$. The proof of typing for $E[a_r]$ must end in the Ty-LetExt rule

$$
\Gamma \vdash [a_i : u_i] \northern \Gamma, v : u \vdash e : t \\
\Gamma \vdash [\textit{"s^n"} : (u_1, \ldots, u_n) \rightarrow u] \northern \Gamma \vdash \textbf{let} \ v : u = ("s^n" : (u_1, \ldots, u_n) \rightarrow u)(a_1, \ldots, a_n) \ \textbf{in} \ e : t
$$

From the Atom Substitution Lemma 6.4.5, and the premise $\Gamma \vdash A[a_r] : u_i$, we know $\Gamma \vdash A[a_c] : u_i$, and we can infer that $\Gamma \vdash E[a_c] : t$. $\triangle$

**TailCall** Suppose $E = A(a_1, \ldots, a_n) : t$. The type proof must use the rule for Ty-TailCall.

$$
\Gamma \vdash A[a_r] : (u_1, \ldots, u_n) \rightarrow \textit{enum}_0 \\
\Gamma \vdash [a_i : u_i] \northern \Gamma \vdash A[a_r](a_1, \ldots, a_n) : \textit{enum}_0
$$

From the premise $\Gamma \vdash A[a_r] : (t_1, \ldots, t_n) \rightarrow t$ we know $\Gamma \vdash A[a_r] : (t_1, \ldots, t_n) \rightarrow t$. From the Atom Substitution Lemma 6.4.5 we know $\Gamma \vdash A[a_c] : (t_1, \ldots, t_n) \rightarrow t$, and we can infer that $\Gamma \vdash E[a_c] : t$. Similarly, if $E = v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_{n-1})$, then $\Gamma \vdash A[a_r] : u_i$. We can infer that $\Gamma \vdash A[a_c] : u_i$, and conclude that $\Gamma \vdash E[a_c] : t$. $\triangle$

**SpecialCall** Suppose $E = \textbf{special} S$. The type proof must use the rule for Ty-Special-Call.

$$
\Gamma \vdash S[a_r] : \textit{special} t \\
\Gamma \vdash \textit{special} S[a_r] : t
$$

From the premise $S[a_r] : \textit{special} t$ and the SpecialCall Substitution Lemma 6.4.6, we know $S[a_c] : \textit{special} t$, from with we infer that $\Gamma \vdash E[a_c] : t$. $\triangle$

**Match** Suppose $E = \textbf{match} A \ \textbf{with} \ [s_i \mapsto e_i] \northern$. The typing proof must use one of the typing inferences for match expressions. Suppose the proof ends with the rule for Ty-Match-Int.

$$
[s_i \in \textit{set}_i] \northern \bigcup_{i=1}^n s_i = \mathbb{Z}_{31} \\
\Gamma \vdash A[a_r] : \mathbb{Z}_{31} \\
\Gamma \vdash [e_i : t] \northern
$$

From the premise $\Gamma \vdash A[a_r] : \mathbb{Z}_{31}$, we can infer that $\Gamma \vdash A[a_c] : \mathbb{Z}_{31}$, and conclude that $\Gamma \vdash E[a_c] : t$. The remaining match cases are similar. $\triangle$
**Alloc** Suppose $E = \text{let } v = L \text{ in } e$. The typing proof must use the $\text{Ty-Alloc}$ rule.

$$\begin{align*}
\hat{\Gamma} &\vdash L[a_r] : \text{alloc} u \\
\hat{\Gamma}, v : u + e : t &\vdash \text{let } v = L[a_r] \text{ in } e : t
\end{align*}$$

From the premise $\Gamma \vdash L[a_r] : \text{alloc} u$ and the Allocation Substitution Lemma 6.4.7, we can conclude $\Gamma \vdash L[a_r] : \text{alloc} u$, and infer that $\Gamma \vdash E[a_r] : t$. \(\triangle\)

**LetSubscript** Suppose $E$ is one of the following.

$$E ::= \begin{align*}
\text{let } &v : t = A[a_2] \text{ in } e \\
\text{let } &v : t = h[A] \text{ in } e
\end{align*}$$

The type judgment must end with one of the rules $\text{Ty-LetSub-RawData}$, $\text{Ty-LetSub-Array}$, $\text{Ty-LetSub-Tuple}$, $\text{Ty-LetSub-Union}$, or $\text{Ty-LetSub-Frame}$. The cases are similar; we illustrate with the case for arrays.

$$\begin{align*}
\hat{\Gamma} &\vdash a_1 : u \text{ array} \\
\hat{\Gamma} &\vdash a_2 : \text{offset} \\
\hat{\Gamma}, v : u + e : t &\vdash \text{let } v = a_1[a_2] \text{ in } e : t
\end{align*}$$

We have the following cases for the type judgment $\Gamma \vdash A[a_r] : t_A$.

- If $A[a_r] = a_1$, then $t_A = u \text{ array}$.
- If $A[a_r] = a_2$, then $t_A = \text{offset}$.

From the Atom Substitution Lemma 6.4.5, we know $\Gamma \vdash A[a_r] : t_A$, and we can infer that $\Gamma \vdash E[a_r] : t$. \(\triangle\)

**SetSubscript** Suppose $E$ is one of the following.

$$E ::= \begin{align*}
A[a_2] : t &\leftarrow a_3 ; e \\
v[A] : t &\leftarrow a_3 ; e \\
v[h] : t &\leftarrow A ; e
\end{align*}$$

The type judgment must end with one of the typing rules $\text{Ty-SetSub-RawData}$, $\text{Ty-SetSub-Array}$, $\text{Ty-SetSub-Tuple}$, $\text{Ty-SetSub-Union}$, or $\text{Ty-SetSub-Frame}$. The cases are similar; we illustrate with the case for arrays.

$$\begin{align*}
\hat{\Gamma} &\vdash a_1 : u \text{ array} \\
\hat{\Gamma} &\vdash a_2 : \text{offset} \\
\hat{\Gamma} &\vdash a_3 : u \\
\hat{\Gamma} &\vdash e : t
\end{align*}$$

We have the following cases for the type judgment $\Gamma \vdash A[a_r] : t_A$.

- If $A[a_r] = a_1$, then $t_A = u \text{ array}$.
- If $A[a_r] = a_2$, then $t_A = \text{offset}$.
- If $A[a_r] = a_3$, then $t_A = u$.

From the Atom Substitution Lemma 6.4.5, we know $\Gamma \vdash A[a_r] : t_A$, and we can infer that $\Gamma \vdash E[a_r] : t$. \(\triangle\)
SetGlobal Suppose \( E = \text{global} : t \leftarrow A; e \). The proof of typing must end with Ty-SetGlobal.

\[
\begin{align*}
\hat{\Gamma} \vdash & \; u & \quad \hat{\Gamma} \vdash & \; A[a_r] : u & \quad \hat{\Gamma} \vdash & \; e : t \\
\hat{\Gamma} \vdash & \; \text{global} : u \leftarrow A[a_r]; e : t
\end{align*}
\]

From the premise \( A[a_r] : u \) and the Atom Substitution Lemma 6.4.5, we know \( \Gamma \vdash A[a_r] : u \), and we can infer that \( \Gamma \vdash E[a_r] : t \). \( \triangle \)

\( \square \)

6.5 Type substitution

The final lemma needed for the preservation and progress properties is the type substitution lemma, which provides the relation between type definitions in the environment and type substitution. The proof is a straightforward structural induction on the type \( t_r \).

Lemma 6.5.1 Type Substitution The following rule is derivable.

\[
\begin{align*}
\Gamma, \alpha : \Omega = u_r \vdash & \; t_r = t_r : \Omega \\
\hat{\Gamma}, \alpha : \Omega = u_r \vdash & \; t_r[a_r/\alpha] = t_r : \Omega \\
\text{Type-Subst}
\end{align*}
\]

We prove this by structural induction on the type \( t_r \).

Constant types If \( t_r \) is any of \( \text{Z}_{\Omega}, \text{enum}_i, \mathbb{Z}_{\text{pre}}^\text{sign}, \mathbb{R}_{\text{pre}}, \text{data}, v\.i \), or a type variable \( \beta \neq \alpha \), the substitution has no effect, and the result follows trivially. \( \triangle \)

Type variables Suppose \( t_r = \alpha \). To prove \( u_r = \alpha : \Omega \), we apply the Ty-Apply-2 rule (after applying the Ty-Sym rule).

\[
\begin{align*}
\Gamma \vdash & \; u_r = u_r : \Omega & \quad \Gamma \vdash & \; \alpha = \alpha : \Omega \\
\hat{\Gamma}, \alpha : \Omega = u_r \vdash & \; \alpha = u_r : \Omega \\
\text{(\( \hat{\Gamma}', \alpha : \Omega = u_r \)) as \( \hat{\Gamma} \vdash \alpha = u_r : \Omega \)}
\end{align*}
\]

The premise \( \Gamma \vdash u_r = u_r : \Omega \) follows from the Well-Formed Context Values Lemma 6.1.4. Since \( t_r = \alpha \), it follows that \( \Gamma \vdash \alpha = \alpha : \Omega \) from the premise of Type-Subst. \( \triangle \)

Tuples Suppose \( t_r = \langle t_1, \ldots, t_n \rangle_{\text{tuple, class}} \). The proof that \( t_r \) is a well-formed type must use one of the rules WF-ST-TyTuple-Normal or WF-ST-TyTuple-Boxed. We illustrate this proof case with the WF-ST-TyTuple-Normal; the proofs are similar.

\[
\begin{align*}
\hat{\Gamma} \vdash & \; [t_1 = t_i : \omega]_1^n \\
\hat{\Gamma} \vdash & \; \emptyset \\
\hat{\Gamma} \vdash & \; \langle t_1, \ldots, t_n \rangle_{\text{normal}} = \langle t_1, \ldots, t_n \rangle_{\text{normal}} : \omega
\end{align*}
\]

From the premise \([t_i = t_i : \omega]_1^n\), we can assume by induction that \( [t_i[u_r/\alpha] = t_i : \omega]_1^n \).
We can then re-apply the \textsc{WF-ST-TyTuple-Normal}.

\[
\Gamma \vdash [t_i[u_r/\alpha] = t_i : \omega]_i^n \quad \Gamma \vdash \phi \\
\Gamma \vdash \langle t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha]\rangle_{\text{normal}} = \langle t_1, \ldots, t_n\rangle_{\text{normal}} : \omega
\]

\[
\triangle
\]

**Arrays** Suppose \( t_r = \text{array} \). The proof that \( t_r \) is well-formed must use the rule \textsc{WF-ST-TyArray}.

\[
\begin{align*}
\hat{\Gamma} & \vdash t = \Omega \\
\hat{\Gamma} & \vdash \text{array} = \text{array} : \omega
\end{align*}
\]

From the premise \( t = t : \omega \), we can assume by induction that \( t[u_r/\alpha] = t : \omega \).

We can then re-apply the \textsc{WF-ST-TyArray} rule.

\[
\begin{align*}
\Gamma & \vdash t[u_r/\alpha] = t : \Omega \\
\Gamma & \vdash t[u_r/\alpha] \text{ array} = t \text{ array} : \omega
\end{align*}
\]

\[
\triangle
\]

**Unions** Suppose \( t = \text{union}(tv[t_1, \ldots, t_n], set_I) \). The proof that \( t_r \) is a well-formed type must use the rule \textsc{WF-ST-TyUnion}.

\[
\begin{align*}
\text{set}_I \subseteq \{0 \ldots n - 1\} & \quad \hat{\Gamma} \vdash [t_i = t_i : \omega]_i^n \\
\hat{\Gamma} & \vdash tv[t_1, \ldots, t_m] : \omega_{\text{union}[n]} \\
\hat{\Gamma} & \vdash \text{union}(tv[t_1, \ldots, t_m], set_I) = \text{union}(tv[t_1, \ldots, t_m], set_I) : \omega
\end{align*}
\]

From the premise \( [t_i = t_i : \omega]_i^n \), we can assume by induction that \( [t_i[u_r/\alpha] = t_i : \omega]_i^n \).

We can then re-apply the \textsc{WF-ST-TyUnion} rule.

\[
\begin{align*}
\text{set}_I \subseteq \{0 \ldots n - 1\} & \quad \Gamma \vdash [t_i[u_r/\alpha] = t_i : \omega]_i^n \\
\Gamma & \vdash tv[t_1, \ldots, t_m] : \omega_{\text{union}[n]} \\
\Gamma & \vdash \text{union}(tv[t_1[u_r/\alpha], \ldots, t_m[u_r/\alpha]], set_I) = \text{union}(tv[t_1, \ldots, t_m], set_I) : \omega
\end{align*}
\]

\[
\triangle
\]

**Frames** Suppose \( t_r = \text{frame}(tv[t_1, \ldots, t_n]) \). The proof that \( t_r \) is a well-formed type must use the rule \textsc{WF-ST-TyFrame}.

\[
\begin{align*}
\hat{\Gamma} & \vdash [t_i = t_i : \omega]_i^n \\
\hat{\Gamma} & \vdash tv[t_1, \ldots, t_m] : \omega_{\text{frame}} \\
\Gamma & \vdash \text{frame}(tv[t_1, \ldots, t_m]) = \text{frame}(tv[t_1, \ldots, t_m]) : \omega
\end{align*}
\]

From the premise \( [t_i = t_i : \omega]_i^n \), we can assume by induction that \( [t_i[u_r/\alpha] = t_i : \omega]_i^n \).

We can then re-apply the \textsc{WF-ST-TyFrame} rule.

\[
\begin{align*}
\Gamma & \vdash [t_i[u_r/\alpha] = t_i : \omega]_i^n \\
\Gamma & \vdash tv[t_1, \ldots, t_m] : \omega_{\text{frame}} \\
\Gamma & \vdash \text{frame}(tv[t_1[u_r/\alpha], \ldots, t_m[u_r/\alpha]]) = \text{frame}(tv[t_1, \ldots, t_m]) : \omega
\end{align*}
\]

\[
\triangle
\]
Type application Suppose \( t_r = tv[t_1, \ldots, t_n] \). The proof that \( t_r \) is a well-formed type must use the rule \( \text{TY-APPLY-1} \).

\[
\begin{align*}
\hat{\Gamma} \vdash [t_i = t_i : \omega]_1^m & \quad \hat{\Gamma} \vdash tv : \omega^m \rightarrow k_s \\
\hat{\Gamma} \vdash tv[t_1, \ldots, t_m] = tv[t_1, \ldots, t_m] : k_s
\end{align*}
\]

From the premise \( [t_i = t_i : \omega]_1^n \), we can assume by induction that \( [t_i[u_r/\alpha] = t_i : \omega]_1^n \).
We can then re-apply the \( \text{TY-APPLY-1} \) rule.

\[
\begin{align*}
\hat{\Gamma} \vdash [t_i[u_r/\alpha] = t_i : \omega]_1^m & \quad \hat{\Gamma} \vdash tv : \omega^m \rightarrow k_s \\
\hat{\Gamma} \vdash tv[t_1[u_r/\alpha], \ldots, t_m[u_r/\alpha]] = tv[t_1, \ldots, t_m] : k_s
\end{align*}
\]

\( \triangle \)

Functions Suppose \( t_r = (t_1, \ldots, t_n) \rightarrow u \). The proof that \( t_r \) is well-formed must end with the rule \( \text{WF-ST-TyFun-Mono} \).

\[
\begin{align*}
\hat{\Gamma} \vdash [t_i = t_i : \Omega]_1^m & \quad \hat{\Gamma} \vdash u = u : \Omega \\
\hat{\Gamma} \vdash ((t_1, \ldots, t_n) \rightarrow u) = ((t_1, \ldots, t_n) \rightarrow u) : \omega
\end{align*}
\]

From the premises \( [t_i = t_i : \omega]_1^n \), and \( u = u : \Omega \), we can assume by induction that \( [t_i[u_r/\alpha] = t_i : \omega]_1^n \) and \( u[u_r/\alpha] = u : \Omega \). We can then re-apply the \( \text{WF-ST-TyFun-Mono} \) rule.

\[
\begin{align*}
\hat{\Gamma} \vdash [t_i[u_r/\alpha] = t_i : \Omega]_1^m & \quad \hat{\Gamma} \vdash u[u_r/\alpha] = u : \Omega \\
\hat{\Gamma} \vdash ((t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha]) \rightarrow u[u_r/\alpha]) = ((t_1, \ldots, t_n) \rightarrow u) : \omega
\end{align*}
\]

\( \triangle \)

Universal quantification Suppose \( t_r = \forall \alpha_1, \ldots, \alpha_n. t \). Then \( t_r \) must be a function type \( (t_1, \ldots, t_n) \rightarrow u \), and the proof that \( t_r \) is well-formed must end with the rule \( \text{WF-ST-TyFun-Poly} \).

\[
\begin{align*}
\hat{\Gamma}, [\alpha_i : \omega]_1^m \vdash ((t_1, \ldots, t_n) \rightarrow u) = ((t_1, \ldots, t_n) \rightarrow u) : \omega \\
\hat{\Gamma} \vdash \forall \alpha_1, \ldots, \alpha_m. ((t_1, \ldots, t_n) \rightarrow u) = \forall \alpha_1, \ldots, \alpha_m. ((t_1, \ldots, t_n) \rightarrow u) : \omega
\end{align*}
\]

From the premises \( [t_i = t_i : \omega]_1^n \), and \( u = u : \Omega \), we can assume by induction that \( [t_i[u_r/\alpha] = t_i : \omega]_1^n \) and \( u[u_r/\alpha] = u : \Omega \). We can then re-apply the \( \text{WF-ST-TyFun-Poly} \) rule.

\[
\begin{align*}
\hat{\Gamma}, [\alpha_i : \omega]_1^m \vdash ((t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha]) \rightarrow u[u_r/\alpha]) = ((t_1, \ldots, t_n) \rightarrow u) : \omega \\
\hat{\Gamma} \vdash \forall \alpha_1, \ldots, \alpha_m. ((t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha]) \rightarrow u[u_r/\alpha]) = \forall \alpha_1, \ldots, \alpha_m. ((t_1, \ldots, t_n) \rightarrow u) : \omega
\end{align*}
\]

\( \triangle \)

Existential quantification Suppose \( t_r = \exists \alpha_1, \ldots, \alpha_n. t \). Then the proof that \( t_r \) is well-formed must end with the rule \( \text{WF-ST-TyExists} \).
\[
\frac{\hat{\Pi}, [\alpha_i : \omega]_1^m \vdash t : \Omega}{\Gamma \vdash \exists \alpha_1, \ldots, \alpha_m. t \equiv \exists \alpha_1, \ldots, \alpha_m. t : \omega}
\]

From the premises \([t_i = t_i : \omega]_1^n\), and \(u = u : \Omega\), we can assume by induction that \([t_i[u_r/\alpha] = t_i : \omega]_1^n\). We can then re-apply the WF-ST-TyEXISTS rule.

\[
\frac{\hat{\Pi}, [\alpha_i : \omega]_1^m \vdash t[u_r/\alpha] = t : \Omega}{\Gamma \vdash \exists \alpha_1, \ldots, \alpha_m. t[u_r/\alpha] = \exists \alpha_1, \ldots, \alpha_m. t : \omega}
\]

\(\triangleq\)

This concludes the proof of \(\Gamma, \alpha : \Omega = u_r \vdash t_r = t_r : \Omega\).

\(\square\)

### 6.6 Fully-defined contexts

**Lemma 6.6.1** Fully-defined contexts Assume all of the following are true.

- \(\Gamma_r\) is fully defined.
- \(\Gamma_r \vdash e_r : \text{enum}_0\)
- For each checkpoint \((\Gamma', f(\phi, a_1, \ldots, a_n)) \in \mathcal{C}_r\), the environment \(\Gamma'\) is fully-defined, and \(\Gamma', v_c : \mathbb{Z}_{\text{signed}} \vdash f(v_c, a_1, \ldots, a_n) : \text{enum}_0\).
- \((\Gamma_r | C_r \mid e_r) \rightarrow (\Gamma_c | C_c \mid e_c)\)

Then \(\Gamma_c\) is fully defined, and for each \((\Gamma', f(\phi, a_1, \ldots, a_n)) \in \mathcal{C}_c\), the environment \(\Gamma'\) is fully-defined.

The proof is by a case analysis on the reduction rule. We will consider only those operational rules that modify \(\Gamma_r\) or \(\mathcal{C}_r\), as the other cases are trivial.

**Type application** Suppose \(e_r = E[\text{ty\_apply}[u](f, u_1, \ldots, u_m)]\). The reduction rule is RED-ATOM-APPLY.

\[
((\Gamma'_r, f : u' = \Lambda \alpha_1, \ldots, \alpha_m. b) \text{ as } \Gamma_r | C_r \mid E[\text{ty\_apply}[u](f, u_1, \ldots, u_m)]) \rightarrow
(\Gamma_r, [\alpha_i : \omega = u_i]_1^m, g : u = b \mid C_r \mid E[g])
\]

This rule introduces the new definitions \([\alpha_i : \omega = u_i]_1^m\) and \(g : t = b\).

\(\triangleq\)

**Type abstraction** Suppose \(e_r = E[\text{ty\_pack}[u](v_1, u_1, \ldots, u_m)]\). The operational rule is RED-ATOM-PACK.

\[
((\Gamma'_r, v_1 : u' = b) \text{ as } \Gamma_r | C_r \mid E[\text{ty\_pack}[u](v_1, u_0, \ldots, u_{m-1})]) \rightarrow
(\Gamma_r, v_2 : u = \text{ty\_pack}[u](v_1, u_0, \ldots, u_{m-1}) \mid C_r \mid E[v_2])
\]

This rule introduces the new definition \(v_2 : u = \text{ty\_pack}[u](v_1, u_1, \ldots, u_m)\).

\(\triangleq\)
Type unpacking Suppose \( e_r = E[\text{ty\_unpack}(v_1)] \). The operational rule is Red-Atom-Unpack.

\[
\begin{align*}
(\Gamma'_r, v_1: u & = \text{ty\_pack}[\exists [\alpha_i]_{0}^{m-1}.t([v, [u_i]_{0}^{m-1}]) \text{ as } \Gamma'_r \mid C_r \mid E[\text{ty\_unpack}(v_1)]]) \\
\quad (\Gamma_r, [\alpha_i: \omega = u_i]_{0}^{m-1}, v_2: t = v \mid C_r \mid E[v_2])
\end{align*}
\]

The rule introduces the new definitions \([\alpha_i: \omega = u_i]_{0}^{m-1}\) and \(v_2: t = v\).

LetAtom Suppose \( e_r = \text{let } u \equiv h \text{ in } e \). The operational rule is Red-LetAtom.

\[
(\Gamma_r \mid C_r \mid \text{let } u \equiv h \text{ in } e) \rightarrow (\Gamma_r, v: u \equiv h \mid C_r \mid e)
\]

The rule introduces the new definition \(v: u \equiv h\).

LetExt Suppose \( e_r = \text{let } v: t = ("s" : (u_1, \ldots, u_n) \rightarrow t)(h_1, \ldots, h_n) \text{ in } e \). The operational rule is Red-LetExt.

\[
\begin{align*}
(\Gamma_r \mid C_r \mid \text{let } v: t & = ("s" : (u_1, \ldots, u_n) \rightarrow t)(h_1, \ldots, h_n) \text{ in } e) \\
\quad (\Gamma_r, v: t = \llbracket "s" \rrbracket(h_1, \ldots, h_n) \mid C_r \mid e)
\end{align*}
\]

The rule introduces the new definition \(v: t = \llbracket "s" \rrbracket(h_1, \ldots, h_n)\).

TailCall Suppose \( e_r = v(h_1, \ldots, h_n) \). The operational rule is Red-TailCall.

\[
\begin{align*}
((\Gamma'_r, v: (u_1, \ldots, u_n) \rightarrow t = \lambda v_1, \ldots, v_n. e) \text{ as } \Gamma'_r \mid C_r \mid v(h_1, \ldots, h_n)) & \rightarrow (\Gamma_r \mid C_r \mid f(h_1, \ldots, h_n)) \\
\quad (\Gamma_r, [v_i: u_i = h_i]_1^n \mid C_r \mid e)
\end{align*}
\]

The rule introduces the new definitions \([v_i: u_i = h_i]_1^n\).

Special-calls There are several special-calls.

- For \( e_r = \text{special migrate } [j, h_{\text{ptr}}, h_{\text{off}}] f(h_1, \ldots, h_n) \), the operational rule is Red-SysMigrate.

\[
(\Gamma_r \mid C_r \mid \text{special migrate } [j, h_{\text{ptr}}, h_{\text{off}}] f(h_1, \ldots, h_n)) \rightarrow (\Gamma_r \mid C_r \mid f(h_1, \ldots, h_n))
\]

The contexts are not modified.
- For \( e_r = \text{special atomic } f(h_{\text{const}}, h_1, \ldots, h_n) \), the operational rule is Red-Atomic.

\[
(\Gamma_r \mid C_r \mid \text{special atomic } f(h_{\text{const}}, h_1, \ldots, h_n)) \rightarrow (\Gamma_r \mid C_r \mid f(h_{\text{const}}, h_1, \ldots, h_n))
\]

By assumption the context \( \Gamma_r \) is fully-defined.
- For \( e_r = \text{special rollback } [i, j] \), if \( i \neq 0 \), the operational rule is Red-Atomic-rollback-1.

\[
\begin{align*}
(\Gamma'_r, \Gamma_m; \ldots; C_1 = (\Gamma_r, f(\circ, h_1, \ldots, h_n)); \ldots; C_1 \mid \text{special rollback } [i, j]) \\
\quad (\Gamma_r \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, h_1, \ldots, h_n))
\end{align*}
\]

This rule removes the checkpoints \( C_m, \ldots, C_{i+1} \); the remaining checkpoints are unchanged.
If \( i = 0 \), the operational rule is RED-ATOMIC-ROLLBACK-2, which does not change the checkpoint environment.

\[
(\Gamma'_r | C_m = \langle \Gamma_r, f(o, h_1, \ldots, h_n) \rangle; \ldots; C_i | \text{special rollback } [0, j]) \rightarrow
(\Gamma_r | C_m; \ldots; C_i | f(j, h_1, \ldots, h_n))
\]

- For \( e_r = \text{special commit } \langle i \rangle f(h_1, \ldots, h_n) \), the operational rule is either RED-ATOMIC-COMMIT-1 or RED-ATOMIC-COMMIT-2.

\[
(\Gamma_r | C_m; \ldots; C_i | \text{special commit } \langle i \rangle f(h_1, \ldots, h_n)) \rightarrow
(\Gamma_r | C_m'; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 | f(h_1, \ldots, h_n))
\text{ when } i \in \{1 \ldots m\}
\]

\[
(\Gamma_r | C_m; \ldots; C_1 | \text{special commit } [0] f(h_1, \ldots, h_n)) \rightarrow
(\Gamma_r | C_{m-1}; \ldots; C_1 | f(h_1, \ldots, h_n))
\]

In both cases, a checkpoint is removed from the checkpoint environment \( C_r \), but the remaining checkpoints are not modified.

\( \triangle \)

**Allocation** There are several cases here for each of the allocation types.

- For \( e_r = \text{let } v = \langle h_1, \ldots, h_n \rangle \_ : u \text{ in } e \), the operational rule is RED-ALLOC-TUPLE.

\[
(\Gamma_r | C_r | \text{let } v = \langle h_1, \ldots, h_n \rangle \_ : u \text{ in } e) \rightarrow (\Gamma_r, v: u = \langle h_1, \ldots, h_n \rangle | C_r | e)
\]

The rule introduces the new definition \( v: u = \langle h_1, \ldots, h_n \rangle \).

- For \( e_r = \text{let } v = \text{union}(tv[h_1, \ldots, h_n], j) : t \text{ in } e \), the operational rule is RED-ALLOC-UNION.

\[
(\Gamma_r | C_r | \text{let } v = \text{union}(tv[h_1, \ldots, h_n], j) : t \text{ in } e) \rightarrow (\Gamma_r, v: t = tv(j, \ldots, h_n) | C_r | e)
\]

The rule introduces the new definition \( v: t = tv(j, \ldots, h_n) \).

- For \( e_r = \text{let } v = \text{array}(i_{\text{size}}, h_{\text{init}}) : t \text{ in } e \), the operational rule is RED-ALLOC-ARRAY.

\[
(\Gamma_r | C_r | \text{let } v = \text{array}(i_{\text{size}}, h_{\text{init}}) : t \text{ in } e) \rightarrow (\Gamma_r, v: t = \langle h_{\text{init}}^1, \ldots, h_{\text{init}}^{\text{in}} \rangle | C_r | e)
\text{ when } i_{\text{size}} \geq 0
\]

The rule introduces the new definition \( v: t = \langle h_{\text{init}}^1, \ldots, h_{\text{init}}^{\text{in}} \rangle \).

- For \( e_r = \text{let } v = \text{malloc}(i_{\text{size}}) : t \text{ in } e \), the operational rule is RED-ALLOC-MALLOC.

\[
(\Gamma_r | C_r | \text{let } v = \text{malloc}(i_{\text{size}}) : t \text{ in } e) \rightarrow (\Gamma_r, v: t = \langle c \rangle | C_r | e)
\text{ when } i_{\text{size}} \geq 0
\]

The rule introduces the new definition \( v: t = \langle c \rangle \).

- For \( e_r = \text{let } v = \text{frame}(tv[t_1, \ldots, t_m]) \text{ in } e \), the operational rule is RED-ALLOC-FRAME.

\[
(\Gamma_r | C_r | \text{let } v = \text{frame}(tv[t_1, \ldots, t_m]) \text{ in } e) \rightarrow
(\Gamma_r, v: \text{frame}(tv[t_1, \ldots, t_m]) = \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle | C_r | e)
\]

The rule introduces the new definition \( v: \text{frame}(tv[t_1, \ldots, t_m]) = \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle \).
Subscripting There are several subscript cases for the expression $e_r = \texttt{let} v_1 : t_1 = v_2[i] \texttt{in} e$.

- The operational rule RED-LET-SUB-TUPLE introduces the new definition $v_1 : u = h_j$.

$$(\Gamma'_r, v_2 : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r | C_r | \texttt{let} v_1 : u = v_2[i] \texttt{in} e \rightarrow (\Gamma_r, v_1 : u = h_j | C_r | e)$$

- The operational rule RED-LET-SUB-ARRAY introduces the new definition $v_1 : u = h_j$.

$$(\Gamma'_r, v_2 : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r | C_r | \texttt{let} v_1 : u = v_2[j] \texttt{in} e \rightarrow (\Gamma_r, v_1 : u = h_j | C_r | e) \text{ when } j \in \{0 \ldots n - 1\}$$

- The operational rule RED-LET-SUB-UNION introduces the new definition $v_1 : u = h_k$.

$$(\Gamma'_r, v_2 : u' = tv_j(h_0, \ldots, h_{n-1})) \text{ as } \Gamma_r | C_r | \texttt{let} v_1 : u = v_2[k] \texttt{in} e \rightarrow (\Gamma_r, v_1 : u = h_k | C_r | e)$$

- The operational rule RED-LET-SUB-RawData introduces the new definition $v_1 : u = h$.

$$(\Gamma'_r, v_2 : u' = \langle c \rangle) \text{ as } \Gamma_r | C_r | \texttt{let} v_1 : u = v_2[j] \texttt{in} e \rightarrow (\Gamma_r, v_1 : u = h | C_r | e) \texttt{ when runtime}(\Gamma_r | \langle c \rangle [j] : u) = h$$

- The operational rule RED-LET-SUB-Frame introduces the new definition $v_1 : u = h$.

$$(\Gamma'_r, v_2 : u' = \langle c : \texttt{frame}(tv[t_1, \ldots, t_m]) \rangle) \text{ as } \Gamma_r | C_r | \texttt{let} v_1 : u = v_2[j] \texttt{in} e \rightarrow (\Gamma_r, v_1 : u = h | C_r | e) \texttt{ when runtime}(\Gamma_r | \langle c : \texttt{frame}(tv[t_1, \ldots, t_m])[j] : u) = h$$

Subscript assignment There are several subscript assignment cases for the expression $v_1[i] : t_2 \leftarrow h ; e$.

- RED-SET-SUB-TUPLE

$$(\Gamma'_r, v : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r | C_r | v[j] : u \leftarrow h ; e \rightarrow (\Gamma'_r, v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle ) | C_r | e)$$

- RED-SET-SUB-ARRAY

$$(\Gamma'_r, v : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r | C_r | v[j] : u \leftarrow h ; e \rightarrow (\Gamma'_r, v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle ) | C_r | e) \texttt{ when } j \in \{0 \ldots n - 1\}$$

- RED-SET-SUB-UNION

$$(\Gamma_r, v : u' = tv_j(h_0, \ldots, h_{n-1}) | C_r | v[k] : u \leftarrow h ; e \rightarrow (\Gamma_r, v : u' = tv_j(h_0, \ldots, h_{k-1}, h, h_{k+1}, \ldots, h_{n-1}) | C_r | e)$$
- **RED-SETSub-Raw Data**

\[ ((\Gamma'_r, v; u' = \langle c' \rangle) \text{ as } \Gamma_r \mid C_r \mid v[j]; u \leftarrow h; e) \rightarrow \]

\[ (\Gamma'_r, v; u' = \langle c' \rangle \mid C_r \mid e) \]

\[ \text{when runtime}(\Gamma_r \mid \langle c \rangle[j]; u \leftarrow h) = \langle c' \rangle \]

In each of the cases, the definition in the environment is modified, but remains a definition. \(\triangle\)

**LetGlobal** Suppose \(e_r = \text{let } v: u = \text{global } l \text{ in } e\). The operational rule is **RED-LETGLOBAL**.

\[ ((\Gamma'_r, l; u = h) \text{ as } \Gamma_r \mid C_r \mid \text{let } v: u = \text{global } l \text{ in } e) \rightarrow (\Gamma_r, v; u = h \mid C_r \mid e) \]

The rule adds the definition \(v: u = h\) to the context. \(\triangle\)

**SetGlobal** Suppose \(e_r = \text{global } l: u \leftarrow h; e\). The operational rule is **RED-SETGLOBAL**.

\[ ((\Gamma'_r, l; u = h') \text{ as } \Gamma_r \mid C_r \mid \text{global } l: u \leftarrow h; e) \rightarrow (\Gamma'_r, l; u = h \mid C_r \mid e) \]

The definition \(l; u' = h'\) is modified but remains a definition. \(\triangle\)

This concludes the proof of the **Fully-defined contexts** Lemma 6.6.1. \(\square\)

### 6.7 Progress lemmas

The following lemmas are technical lemmas used in the proof of Progress. Most often, they establish the kinds of values that can be associated with each type.

**Lemma 6.7.1 Variable declarations** If \(\Gamma \vdash v : t\), then \(v \in \text{dom}(\Gamma)\).

The only rule that can be used to derive a type for a variable is the **Ty-AtomVar** rule.

\[
\begin{array}{c}
\Gamma, v : t \vdash \phi \\
\hline
\Gamma, v : t \vdash v : t
\end{array}
\]

The following lemma is used in the proof of Progress.

**Lemma 6.7.2 Numeric Values** For any derivation \(\Gamma \vdash h : t\) where the type \(t\) is a numeric type, including \(\mathbb{Z}_{\leq 1}, \text{enum}_n, \mathbb{Z}_{\text{prev}}\), and \(\mathbb{R}_{\text{prev}}\), the value \(h\) is either a constant or a variable.
The cases are similar, we illustrate with the case for integers $\mathbb{Z}_{31}$. There are two rules that can derive $\Gamma \vdash h : \mathbb{Z}_{31}$.

- The first is the Ty-AtomInt rule, where the atom $h$ must be an integer $i$.

\[
\begin{align*}
  i \in \mathbb{Z}_{31}, & \quad \Gamma \vdash \textit{o} \\
  \Gamma & \vdash \textit{int}(i) : \mathbb{Z}_{31}
\end{align*}
\]

- In the second case, the value is derived using the Ty-AtomVar rule.

\[
\begin{align*}
  \Gamma, v : \mathbb{Z}_{31} & \vdash \textit{o} \\
  \Gamma, v & \vdash \mathbb{Z}_{31} \vdash v : \mathbb{Z}_{31}
\end{align*}
\]

In this case, the value is a variable $v$.

\[\square\]

**Lemma 6.7.3** Function Values For any derivation $\Gamma \vdash h : (u_1, \ldots, u_n) \to t$, the value $h$ must be a variable $v$. If $v$ is defined in the context $\Gamma$, the value has the form $\lambda v_1, \ldots, v_n. e$.

The premise $h : (u_1, \ldots, u_n) \to t$ can be proved only with the Ty-AtomVar rule.

\[
\begin{align*}
  \Gamma, v : (u_1, \ldots, u_n) & \to t \vdash \textit{o} \\
  \Gamma, v & \vdash (u_1, \ldots, u_n) \to t \vdash (u_1, \ldots, u_n) \to t
\end{align*}
\]

That is, $h$ must be a variable $v$. The only possible typing rule is the Ty-StoreFunMono rule.

\[
\begin{align*}
  \Gamma, [v_i : u_i]_1^n & \vdash e : t \\
  \Gamma & \vdash \lambda v_1, \ldots, v_n. e : (u_1, \ldots, u_n) \to t
\end{align*}
\]

\[\square\]
Chapter 7

Type safety

The proof of type-safety has two parts. The Preservation theorem 7.1.1 shows that types are preserved during program reduction. The Progress theorem 7.2.1 shows that for well-typed programs, if the expression e being evaluated is not a value, then there is a reduction rule that can be used to evaluate the program one more step.

7.1 Preservation

**Theorem 7.1.1** **Preservation** If \((\Gamma_r | C_r | e_r)\) is a valid program and \((\Gamma_r | C_r | e_r) \rightarrow (\Gamma_c | C_c | e_c)\), then \((\Gamma_c | C_c | e_c)\) is a program.

We prove this case analysis on the reduction. From the Fully-defined Contexts Lemma 6.6.1, we can assume that the contexts in \(C_r\) and the context \(\Gamma_r\) are fully-defined. The preservation proof amounts to showing that types are preserved during reduction.

The proof depends on the type judgment \(\Gamma_r \vdash e_r : \text{enum}\). Type judgments in general require value declarations of the form \(v : t\) rather than value definitions of the form \(v : t = b\). We use the Proof Induction Lemma 6.2.1 throughout this proof to infer use of specific type rules in a partially thinned context \(\Gamma_r\) where some number of thinning rules have been applied.

**Variable reduction** Suppose the reduction uses the rule Red-Atom-Var.

\[(\Gamma_r, v : t_1 = h | C_r | E[v]) \rightarrow (\Gamma_r, v : t_1 = h | C_r | E[h])\]

By assumption \(\Gamma_r, v : t_1 = h \vdash E[v] : \text{enum}_0\). Since \(\Gamma_r, v : t_1 = h \vdash v : t_1\) and \(\Gamma_r, v : t_1 = h \vdash h : t_1\), we can conclude from the Substitution Lemma 6.4.8 that \(\Gamma_r, v : t_1 = h \vdash E[h] : \text{enum}_0\). \(\triangle\)

**Unary operations** Suppose the reduction uses the Red-Atom-Unop rule.

\[(\Gamma_r | C_r | E[\text{unop } \dot{v}]) \rightarrow (\Gamma_r | C_r | E[\text{unop } \dot{v}])\]
CHAPTER 7. TYPE SAFETY

From the assumption $\Gamma_r \vdash E[unop \, i] : \texttt{enum}_0$ and the Expression Typing Lemma 6.4.4, we know that the atom $unop \, i$ has some type $t_a$. The proof of typing must use the TY-ATOM\textsc{UNOP} rule.

\[
\Gamma_r \vdash i : \texttt{arg}(unop) \quad \Gamma_r \vdash \texttt{res}(unop) : \Omega
\]
\[
\Gamma_r \vdash unop \, i : \texttt{res}(unop)
\]

That is, $t_a$ is $\texttt{res}(unop)$. Since the built-in unary operators are well-typed (by assumption), the value $[unop](i)$ also has type $\texttt{res}(unop)$. We can conclude from the Substitution Lemma 6.4.8 that $\Gamma_r \vdash E[[unop](i)] : t_a$. △

Binary operations Suppose the reduction uses the RED-ATOM-BINOP rule.

\[
(\Gamma_r \mid C_r \mid E[i \, \texttt{binop} \, j]) \rightarrow (\Gamma_r \mid C_r \mid E[[\texttt{binop}](i, j)])
\]

From the assumption $\Gamma_r \vdash E[i \, \texttt{binop} \, j] : \texttt{enum}_0$ and the Expression Typing Lemma 6.4.4, we know that the atom $i \, \texttt{binop} \, j$ has some type $t_a$. The proof of typing must use the TY-ATOMBINOP rule.

\[
\Gamma_r \vdash i : \texttt{arg}_1(\texttt{binop}) \quad \Gamma_r \vdash j : \texttt{arg}_2(\texttt{binop}) \quad \Gamma_r \vdash \texttt{res}(\texttt{binop}) : \Omega
\]
\[
\Gamma_r \vdash i \, \texttt{binop} \, j : \texttt{res}(\texttt{binop})
\]

That is, $t_a$ is $\texttt{res}(\texttt{binop})$. Since the built-in binary operators are well-typed (by assumption), the value $[\texttt{binop}](i, j)$ also has type $\texttt{res}(\texttt{binop})$. We can conclude from the Substitution Lemma 6.4.8 that $\Gamma_r \vdash E[[\texttt{binop}](i, j)] : \texttt{enum}_0$. △

Type application Suppose the reduction uses the RED-ATOM-APPLY rule.

\[
((\Gamma_r, f : u' = \Lambda \alpha_1, \ldots, \alpha_m, b) \text{ as } \Gamma_r \mid C_r \mid E[\texttt{ty apply}[u](f, u_1, \ldots, u_m)]) \rightarrow (\Gamma_r, [\alpha_i : \omega = u_i]_1^{m}, g : u = b \mid C_r \mid E[g])
\]

From the assumption $\Gamma_r \vdash E[\texttt{ty apply}[u](v_1, u_1, \ldots, u_m)] : \texttt{enum}_0$, and the Expression Typing Lemma 6.4.4, we know that the atom $\texttt{ty apply}[u](v_1, u_1, \ldots, u_m)$ has some type $t_a$. The proof of typing must use the TY-ATOM\textsc{APPLY} rule.

\[
\Gamma_r \vdash [u_i : \omega]_1^{m} \quad \Gamma_r \vdash v_i : \forall \alpha_1, \ldots, \alpha_m. \, t \quad \Gamma_r \vdash u = t[[u_i/\alpha_i]_1^{m}] : \omega
\]
\[
\Gamma_r \vdash \texttt{ty apply}[u](v_1, u_1, \ldots, u_m) : t[[u_i/\alpha_i]_1^{m}]
\]

That is, $t_a = u = t[[u_i/\alpha_i]_1^{m}]$. By the Type Substitution Lemma 6.5.1, we can infer that

\[
\Gamma_r, \alpha_i : \omega = u_i]_1^{m}, v_2 : t = b \vdash v_2 : t_a
\]

We can conclude from the Substitution Lemma 6.4.8 that $\Gamma_r \vdash E[v_2] : \texttt{enum}_0$. △

Type abstraction Suppose the reduction uses the RED-ATOM-PACK rule.

\[
((\Gamma'_r, v_1 : u' = b) \text{ as } \Gamma_r \mid C_r \mid E[\texttt{ty pack}[u](v_1, u_0, \ldots, u_{m-1})]) \rightarrow (\Gamma_r, v_2 : u = \texttt{ty pack}[u](v_1, u_0, \ldots, u_{m-1}) \mid C_r \mid E[v_2])
\]
From the assumption $\Gamma_r \vdash E[\texttt{ty\_pack}[u][v_1, u_1, \ldots, u_m]] : \texttt{enum}_0$, and the EXPRESSION TYING Lemma 6.4.4, we know that the atom $\texttt{ty\_pack}[u](v_1, u_1, \ldots, u_m)$ has some type $t_a$. The proof of typing must use the TY-ATOM-TYPACK rule.
\[
\Gamma_r \vdash u_i : \omega_1 \quad \Gamma_r \vdash t_i : t[\alpha_i]_0^m \quad \Gamma_r \vdash u = \exists \alpha_1, \ldots, \alpha_m.t : \omega
\]
\[
\Gamma_r \vdash \texttt{ty\_pack}[u](v_1, u_1, \ldots, u_m) : \exists \alpha_1, \ldots, \alpha_m.t
\]

That is, $t_a = u = \exists \alpha_1, \ldots, \alpha_m.t$. Since $v_2$ also has type $t_a$, we can infer the from the SUBSTITUTION Lemma 6.4.8 that $\Gamma_r \vdash E[v_2] : \texttt{enum}_0$.

**Type unpacking** Suppose the reduction uses the RED-ATOM-UNPACK rule.
\[
\left(\Gamma_r, v_1 : u = \texttt{ty\_pack}[\exists \alpha_i^{n-1}.t](v, [u_i]_0^{n-1}) \quad \text{as } \Gamma_r \vdash C_r \vdash E[\texttt{ty\_unpack}(v_1)]\right) \rightarrow
\]
\[
\left(\Gamma_r, [\omega = u_i]_0^n, v_2 : t = v \mid C_r \vdash E[v_2]\right)
\]

From the assumption $\Gamma_r \vdash E[\texttt{ty\_unpack}(v_1)] : t$, and the EXPRESSION TYING Lemma 6.4.4, we know that the atom $\texttt{ty\_unpack}(v_1)$ has some type $t_a$. The proof of typing must use the TY-ATOM-TYUNPACK rule.
\[
\Gamma_r \vdash v_1 : \exists \alpha_0, \ldots, \alpha_{m-1}.t
\]
\[
\Gamma_r \vdash \texttt{ty\_unpack}(v_1) : t[\alpha_i]_0^{n-1}
\]

That is, $t_a = t[\alpha_i]_0^{n-1}$. Since $v_1, i = u_i$ for each $i \in \{0 \ldots n-1\}$, we can infer from the TYPE SUBSTITUTION Lemma 6.3.1 that $\Gamma_r \vdash E[v_2] : \texttt{enum}_0$.

**LetAtom** Suppose the reduction uses the rule RED-LET-ATOM.
\[
(\Gamma_r \vdash C_r \mid \texttt{let } v : u = h \texttt{ in } e) \rightarrow (\Gamma_r, v : u = h \mid C_r \vdash e)
\]

The typing proof must use the rule TY-LET-ATOM.
\[
\Gamma_r \vdash h : u \quad \Gamma_r, v : u \vdash e : \texttt{enum}_0
\]
\[
\Gamma_r \vdash \texttt{let } v : u = h \texttt{ in } e : \texttt{enum}_0
\]

Using the THIN-VAR-DEF rule and the premises, we can infer $\Gamma_r, v : t_1 = h \vdash e_r : \texttt{enum}_0$.
\[
\Gamma_r, v : u \vdash h : u \quad \Gamma_r, v : u \vdash e : \texttt{enum}_0
\]
\[
\Gamma_r, v : u = h \vdash e : \texttt{enum}_0
\]

**LetExt** Suppose the reduction uses the rule RED-LET-EXT.
\[
(\Gamma_r \mid C_r \mid \texttt{let } v : t = \texttt{"s" } : (u_1, \ldots, u_n) \rightarrow t)(h_1, \ldots, h_n) \texttt{ in } e) \rightarrow
\]
\[
(\Gamma_r, v : t = \texttt{"s" } (h_1, \ldots, h_n) \mid C_r \mid e)
\]

---

81
The typing proof must use the rule Ty-LetExt.

\[
\Gamma_r, v : u + e : \text{enum}_0 \quad \Gamma_r, [s^n] : (u_1, \ldots, u_n) \rightarrow u
\]

\[
\Gamma_r \vdash \text{let } v = (\text{"s"} : (u_1, \ldots, u_n) \rightarrow u)(h_1, \ldots, h_n) \text{ in } e : \text{enum}_0
\]

From the premises \(\Gamma_r, [h : u_i]^n\) and \(\Gamma_r, [s^n] : (u_1, \ldots, u_n) \rightarrow u\), we can infer that \(\Gamma_r, [s^n](h_1, \ldots, h_n) : u\). Using the Thin-Var-Def rule, we can infer that \(\Gamma_r, v : u = [s^n](h_1, \ldots, h_n) \vdash e : \text{enum}_0\).

\[
\Gamma_r, v : u \vdash [s^n](h_1, \ldots, h_n) : u \quad \Gamma_r, v : u + e : \text{enum}_0
\]

\[
\Gamma_r, v : u = [s^n](h_1, \ldots, h_n) \vdash e : \text{enum}_0
\]

\(\triangle\)

TailCall Suppose the reduction uses the rule Red-TailCall.

\[
((\Gamma_r, v : (u_1, \ldots, u_n) \rightarrow t = \lambda v_1, \ldots, v_n.e) \text{ as } \Gamma_r \mid C_r \mid v(h_1, \ldots, h_n)) \Rightarrow
\]

\[
(\Gamma_r, [v_i : u_i = h_i]^n \mid C_r \mid e)
\]

The proof of typing must use the rule Ty-TailCall.

\[
\Gamma_r \vdash t : \text{enum}_0
\]

\[
\Gamma_r \vdash [u_i]_1^n
\]

\[
\Gamma_r \vdash f(h_1, \ldots, h_n) : \text{enum}_0
\]

By the Well-formed Context Values Lemma 6.1.4 we know that the function is well-formed. The proof must use the Ty-StoreFun-Mono rule.

\[
\Gamma_r, [v_i : u_i = h_i]^n + e : \text{enum}_0
\]

\[
\Gamma_r, \lambda v_1, \ldots, v_n.e : (u_1, \ldots, u_n) \rightarrow \text{enum}_0
\]

From the Thin-Var-Def rule, we can conclude that

\[
\Gamma_r, [v_i : u_i = h_i]^n + e : \text{enum}_0.
\]

\(\triangle\)

SpecialCall There are four cases for reducing a special-call.

- Suppose the reduction uses the rule Red-.SysMigrate.

\[
(\Gamma_r \mid C_r \mid \text{special migrate } [j, h_{ptr}, h_{off}] f(h_1, \ldots, h_n)) \Rightarrow (\Gamma_r \mid C_r \mid f(h_1, \ldots, h_n))
\]

The proof of typing must use the rule Ty-SysMigrate.

\[
\Gamma_r \vdash h_{ptr} : \text{data}
\]

\[
\Gamma_r \vdash h_{off} : \mathbb{Z}_{32}^{\text{signed}}
\]

\[
\Gamma_r \vdash [h_i : t_i]^n
\]

\[
\Gamma_r \vdash f(t_1, \ldots, t_n) \rightarrow \text{enum}_0
\]

\[
\Gamma_r \vdash \text{migrate } [j, h_{ptr}, h_{off}] f(h_1, \ldots, h_n) : \text{special enum}_0
\]

This case is similar to a tail-call. From the premises \(\Gamma_r \vdash [h_i : t_i]^n\) and \(\Gamma_r \vdash f(t_1, \ldots, t_n) \rightarrow \text{enum}_0\), we can infer that \(\Gamma_r \vdash f(h_1, \ldots, h_n) : \text{enum}_0\).
7.1. PRESERVATION

- Suppose the reduction uses the rule RED-ATOMIC.

\[
\left( \Gamma_r \mid C_r \mid \text{specialatomic } f(h_{\text{const}}, h_1, \ldots, h_n) \right) \rightarrow \\
\left( \Gamma_r \mid (\Gamma_r, f(\phi, h_1, \ldots, h_n)) ; C_r \mid f(h_{\text{const}}, h_1, \ldots, h_n) \right)
\]

The proof of typing must use the rule TY-ATOMIC.

\[
\Gamma_r \vdash h_{\text{const}} : \mathbb{Z}_{\text{signed}} \quad \Gamma_r \vdash \left\lfloor h_i : t_i \right\rfloor_1^n \quad \Gamma_r \vdash f : \left(\mathbb{Z}_{\text{signed}}, t_1, \ldots, t_n\right) \rightarrow \text{enum}_0 \\
\Gamma_r \vdash \text{atomic } f(h_{\text{const}}, h_1, \ldots, h_n) : \text{specialenum}_0
\]

From the premises \(\Gamma_r \vdash h_{\text{const}} : \mathbb{Z}_{\text{signed}}, \Gamma_r \vdash \left\lfloor h_i : t_i \right\rfloor_1^n\), and \(\Gamma_r \vdash f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0\), we can infer that the checkpoint \(\langle \Gamma_r, f(\phi, h_1, \ldots, h_n) \rangle\) is well-formed, and that \(\Gamma_r \vdash f(h_1, \ldots, h_n) : \text{enum}_0\).

- Suppose the reduction uses the rule RED-ATOMIC-Rollback-1.

\[
\left( \Gamma'_r \mid C_{m+1} \ldots ; C_1 \mid \text{specialrollback } [i, j] \right) \rightarrow \\
\left( \Gamma_r \mid C_1 ; C_{i-1} ; \ldots ; C_1 \mid f(j, h_1, \ldots, h_n) \right)
\]

By assumption, the checkpoint context \(C_r\) is well-formed, and \(\Gamma_r \vdash f(j, h_1, \ldots, h_n) : \text{enum}_0\). Since the reduction only removes checkpoints from the context, the checkpoint context remains well-formed.

The argument for RED-ATOMIC-Rollback-2 is similar.

- Suppose the reduction uses the rule RED-ATOMIC-Commit-1.

\[
\left( \Gamma_r \mid C_{m+1} \ldots ; C_1 \mid \text{specialcommit } [i] f(h_1, \ldots, h_n) \right) \rightarrow \\
\left( \Gamma_r \mid C_{m+1} \ldots ; C_{i+1} ; C_{i-1} ; \ldots ; C_1 \mid f(h_1, \ldots, h_n) \right)
\]

The proof of typing must use the rule TY-AtomicCommit.

\[
\Gamma_r \vdash i : \mathbb{Z}_{\text{signed}} \quad \Gamma_r \vdash \left\lfloor h_i : t_i \right\rfloor_1^n \quad \Gamma_r \vdash f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0 \\
\Gamma_r \vdash \text{commit } [i] f(h_1, \ldots, h_n) : \text{specialenum}_0
\]

From the premises \(\Gamma_r \vdash \left\lfloor h_i : t_i \right\rfloor_1^n\) and \(\Gamma_r \vdash f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0\), we can infer that \(\Gamma_r \vdash f(h_1, \ldots, h_n) : \text{enum}_0\). The argument for RED-ATOMIC-Commit-2 is similar.

\(\triangle\)

**Match** There are two cases, for matching against an integer, and for matching against a union.

- Suppose the reduction uses the rule RED-Match-Int.

\[
\left( \Gamma_r \mid C_r \mid \text{match } i \text{ with } [s_j \mapsto e_j]_{j=1}^n \right) \rightarrow (\Gamma_r \mid C_r \mid e_k) \\
\text{when } (i \in s_k) \land \forall j \in \{1, \ldots, k-1\}. (i \not\in s_j)
\]

The proof of typing must use one of TY-Match-Int, TY-Match-Enum, or TY-Match-RawInt. We illustrate with the TY-Match-Int rule.
\[
[s_j \in \text{set}_l]_{j=1}^n \quad \bigcup_{j=1}^n s_j = \mathbb{Z}_{\geq 1}^{n_l} \quad \Gamma_r \vdash i : \mathbb{Z}_{\geq 1}^{n_l} \quad \Gamma_r \vdash [e_j : \text{enum}_0]_{j=1}^n
\]

\[
\Gamma_r \vdash \text{match } i \text{ with } [s_j \mapsto e_j]_{j=1}^n : \text{enum}_0
\]

We can infer that \( \Gamma_r \vdash e_k : t \) directly from the premise \( \Gamma_r \vdash [e_j : t]_{j=1}^n \).

- Suppose the reduction uses the rule Red-Match-Union.

\[
(\Gamma_r, v : \text{union}(tv[t_1, \ldots, t_m], s) = tv_j(a_1, \ldots, a_n) \mid C_r \mid \text{match } v \text{ with } [s_i \mapsto e_i]_{1}^{l}) \rightarrow
\]

\[
(\Gamma_r, v : \text{union}(tv[t_1, \ldots, t_m], s_k) = tv_j(a_1, \ldots, a_n) \mid C_r \mid e_k)
\]

\[
\text{when } (j \in s_k) \land \forall p \in \{1, \ldots, k-1\} (i \not\in s_p)
\]

The only possible typing rule for this expression is Ty-Match-Union.

\[
[s_j \in \text{set}_l]_{j=1}^n \{ \} \neq s_{\text{union}} = \bigcup_{j=1}^n s_j \quad \Gamma_r, v : \text{union}(tv[t_1, \ldots, t_m], s_{\text{union}}) \vdash [s_j \mapsto e_j]_{j=1}^n : \text{enum}_0
\]

We can infer that \( \Gamma_r, v : \text{union}(tv[t_1, \ldots, t_m], s_k) = tv_j(a_1, \ldots, a_n) \vdash e_k : \text{enum}_0 \) directly from the premises.

\[\triangle\]

**LetAlloc** There are several allocation cases. In each of these cases, the type judgment uses the Ty-Alloc rule.

\[
\Gamma_r \vdash \text{alloc} : \text{alloc} u \quad \Gamma_r, v : u \vdash e : \text{enum}_0
\]

\[
\Gamma_r \vdash \text{let } v = \text{alloc in } e : \text{enum}_0
\]

- Suppose the reduction uses the rule Red-Alloc-Tuple.

\[
(\Gamma_r \mid C_r \mid \text{let } v = \langle h_1, \ldots, h_n \rangle : u \text{ in } e) \rightarrow (\Gamma_r, v : u = \langle h_1, \ldots, h_n \rangle \mid C_r \mid e)
\]

The corresponding type judgment is Ty-Alloc-Tuple.

\[
\Gamma_r \vdash [h_i : u_i]_{1}^n \quad \Gamma_r \vdash \langle u_1, \ldots, u_n \rangle : \omega
\]

\[
\Gamma_r \vdash \langle h_1, \ldots, h_n \rangle : \text{alloc} \langle u_1, \ldots, u_n \rangle
\]

From the premises \( \Gamma_r \vdash [h_i : u_i]_{1}^n \) we can infer that the \( u = \langle u_1, \ldots, u_n \rangle_{\text{tuple-class}} \) and the context \( \Gamma_r, v : u = \langle h_1, \ldots, h_n \rangle \) is well-formed. From the premise \( e : \text{enum}_0 \) we conclude that \( \Gamma_r, v : u = \langle h_1, \ldots, h_n \rangle \vdash e : \text{enum}_0 \).

- Suppose the reduction uses the rule Red-Alloc-Array.

\[
(\Gamma_r \mid C_r \mid \text{let } v = \text{array}(i_{\text{size}}, h_{in}) : \text{enum}_0 \text{ in } e) \rightarrow (\Gamma_r, v : \text{enum}_0 = \langle h_{in}^1, \ldots, h_{in}^{i_{size}} \rangle \mid C_r \mid e)
\]

\[
\text{when } i_{\text{size}} \geq 0
\]

The corresponding type judgment is Ty-Alloc-Array.
\[
\frac{\Gamma_r \vdash i_{\text{size}} : \text{offset} \quad \Gamma_r \vdash h_{\text{init}} : u'}{
\Gamma_r \vdash \text{array}(i_{\text{size}}, h_{\text{init}}) : \text{alloc} u' \text{ array}}
\]

That is, \( u = u' \text{ array} \). From the premise \( \Gamma_r \vdash h_{\text{init}} : u' \) we can infer that the context \( \Gamma_r, v : u = \langle h_1, \ldots, h_n \rangle \) is well-formed. From the premise \( e : \text{enum}_0 \) we conclude that \( \Gamma_r, v : u = \langle h_1, \ldots, h_n \rangle + e : \text{enum}_0 \).

- Suppose the reduction uses the rule \text{RED-ALLOC-UNION}.

\[\begin{align*}
(\Gamma_r \mid C_r \mid \text{let } v = \text{union}(tv[h_1, \ldots, h_n], j) : t \text{ in } e) &\rightarrow (\Gamma_r, v : t = tv(j, h_1, \ldots, h_n) \mid C_r \mid e)
\end{align*}\]

The corresponding type judgment is \text{TY-ALLOC-UNION}.

\[\begin{align*}
\Gamma_r \vdash tv(a_1, \ldots, a_k) : \text{union}(tv[t_1, \ldots, t_m], \{j\}) &\rightarrow \\
\Gamma_r \vdash \text{union}(tv[a_1, \ldots, a_k, j] : \text{alloc union}(tv[t_1, \ldots, t_m], \{j\})
\end{align*}\]

That is, \( t = \text{union}(tv[t_1, \ldots, t_m], \{j\}) \). From the premise of the typing rule, we can infer that \( \Gamma_r, v : t = tv_i(h_1, \ldots, h_n) \) is a well-formed context, and from the premise \( e : \text{enum}_0 \) we conclude that \( \Gamma_r, v : t = tv_i(h_1, \ldots, h_n) + e : \text{enum}_0 \).

- Suppose the reduction uses the rule \text{RED-ALLOC-MALLOC}.

\[\begin{align*}
(\Gamma_r \mid C_r \mid \text{let } v = \text{malloc}(i_{\text{size}}) : t \text{ in } e) &\rightarrow (\Gamma_r, v : t = \langle c \rangle \mid C_r \mid e)
\end{align*}\]

\( \text{when } i_{\text{size}} \geq 0 \)

The corresponding type judgment is \text{TY-ALLOC-MALLOC}.

\[\begin{align*}
\Gamma_r \vdash h : \text{offset} &\rightarrow \\
\Gamma_r \vdash \text{malloc}(h) : \text{alloc data}
\end{align*}\]

That is \( t = \text{data} \), and we can infer that the context \( \Gamma_r, v : t = \langle c \rangle \) is well-formed. From the premise \( e : \text{enum}_0 \) we conclude that \( \Gamma_r, v : t = \langle c \rangle + e : \text{enum}_0 \).

- Suppose the reduction uses the rule \text{RED-ALLOC-FRAME}.

\[\begin{align*}
(\Gamma_r \mid C_r \mid \text{let } v = \text{frame}(tv[t_1, \ldots, t_m]) \text{ in } e) &\rightarrow \\
(\Gamma_r, v : \text{frame}(tv[t_1, \ldots, t_m]) = \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle \mid C_r \mid e)
\end{align*}\]

The corresponding type judgment is \text{TY-ALLOC-FRAME}.

\[\begin{align*}
\Gamma_r \vdash \text{frame}(tv[u_1, \ldots, u_m]) : \omega &\rightarrow \\
\Gamma_r \vdash \text{alloc frame}(tv[u_1, \ldots, u_m]) : \text{alloc frame}(tv[u_1, \ldots, u_m])
\end{align*}\]

The premise of the typing rule requires that \( tv[t_1, \ldots, t_m] : \omega_{\text{frame}} \). From the \text{TY-STOREFRAME} rule, we can infer that the context \( \Gamma_r, v : \text{frame}(tv[t_1, \ldots, t_m]) = \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle \) is well-formed.

\[\begin{align*}
\Gamma_r \vdash tv[t_1, \ldots, t_m] : \omega_{\text{frame}} &\rightarrow \\
\Gamma_r \vdash \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle : \text{frame}(tv[t_1, \ldots, t_m])
\end{align*}\]

From the premise \( e : \text{enum}_0 \), we conclude that \( \Gamma_r, v : \text{frame}(tv[t_1]^m) = \langle c : \text{frame}(tv[t_1]^m) \rangle + e : \text{enum}_0 \).
LetSubscript There are several subscripting cases.

- Suppose the reduction uses the rule Red-LetSub-Tuple.

\[(\Gamma', v_2 : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r \mid C_r \mid \text{let } v_1 : u = v_2[j] \text{ in } e \rightarrow \]

\[(\Gamma_r, v_1 : u = h_j \mid C_r \mid e) \]

The proof of typing must end in the rule Ty-LetSub-Tuple:

\[\widehat{\Gamma_r} \vdash v_2 : \langle a_0, \ldots, a_{j-1}, u, a_{j+1}, \ldots, a_{n-1} \rangle_c \]

\[\widehat{\Gamma_r}, v_1 : u + e : \text{enum}_0 \]

\[\widehat{\Gamma_r} \vdash \text{let } v : u = v_2[j] \text{ in } e : \text{enum}_0 \]

From the premise \(\Gamma_r \vdash v_2 : \langle a_0, \ldots, a_{j-1}, u, a_{j+1}, \ldots, a_{n-1} \rangle_c\), we can infer that \(u'\) is the tuple type \(\langle a_0, \ldots, a_{j-1}, u, a_{j+1}, \ldots, a_{n-1} \rangle_c\). In addition, the Well-Formed Context Values Lemma 6.1.4 implies that \(\Gamma_r \vdash h_j : u\). Finally, from the premise \(\Gamma_r, v_1 : u + e : \text{enum}_0\) and the Thin-Var-Def rule, we can infer that \(\Gamma_r, v_1 : u = h_j + e : \text{enum}_0\).

- Suppose the reduction uses the rule Red-LetSub-Array.

\[(\Gamma', v_2 : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r \mid C_r \mid \text{let } v_1 : u = v_2[j] \text{ in } e \rightarrow \]

\[(\Gamma_r, v_1 : u = h_j \mid C_r \mid e) \]

when \(j \in \{0, \ldots, n - 1\}\)

The corresponding type rule is Ty-LetSub-Array.

\[\widehat{\Gamma_r} \vdash v_2 : u \text{ array} \]

\[\widehat{\Gamma_r} \vdash j : \text{offset} \]

\[\widehat{\Gamma_r}, v_1 : u + e : \text{enum}_0 \]

\[\widehat{\Gamma_r} \vdash \text{let } v : u = v_2[j] \text{ in } e : \text{enum}_0 \]

From the premise \(\widehat{\Gamma_r} \vdash v_2 : u \text{ array}\), we can infer that \(u' = u \text{ array}\). In addition, the Well-Formed Context Values Lemma 6.1.4 implies that \(\Gamma_r \vdash h_j : u\). Finally, from the premise \(\Gamma_r, v_1 : u + e : \text{enum}_0\) and the Thin-Var-Def rule, we can infer that \(\Gamma_r, v_1 : u = h_j + e : \text{enum}_0\).

- Suppose the reduction uses the rule Red-LetSub-Union.

\[(\Gamma', v_2 : u' = tv_j(h_0, \ldots, h_{n-1})) \text{ as } \Gamma_r \mid C_r \mid \text{let } v_1 : u = v_2[k] \text{ in } e \rightarrow \]

\[(\Gamma_r, v_1 : u = h_k \mid C_r \mid e) \]

The corresponding type rule is Ty-LetSub-Union.

\[\widehat{\Gamma_r} \vdash v_2 : \text{union}(tv[t_1', \ldots, t_m'], \{j\}) \]

\[\widehat{\Gamma_r} \vdash tv[t_1', \ldots, t_m'] = (\overline{u[k]} \cdot 0^{k-1} + \overline{u[k]}_{0}^{n-1}, u, [u[k-1]]_{0}^{n-1} + [u[k+1]]_{0}^{n-1}) : \omega_{\text{union}[n]} \]

\[\widehat{\Gamma_r}, v_1 : u + e : \text{enum}_0 \]

\[\widehat{\Gamma_r} \vdash \text{let } v : u = v_2[k] \text{ in } e : \text{enum}_0 \]

From the premise \(\widehat{\Gamma_r} \vdash v_2 : \text{union}(tv[t_1', \ldots, t_m'], \{j\})\) we infer that \(u' = \text{union}(tv[t_1', \ldots, t_m'], \{j\})\). In addition, the Well-Formed Context Values Lemma 6.1.4 implies that \(\Gamma_r \vdash h_k : u\). Finally, from the premise \(\Gamma_r, v_1 : u + e : \text{enum}_0\) and the Thin-Var-Def rule, we can infer that \(\Gamma_r, v_1 : u = h_j + e : \text{enum}_0\).
7.1. PRESERVATION

- Suppose the reduction uses the rule **RED-LETSUB-RAWDATA**.

\[
\begin{align*}
((\Gamma_r, v_2 : u' = \langle c \rangle) \text{ as } \Gamma_r | C_r \mid \text{let } v_1 : u = v_2[j] \text{ in } e) & \rightarrow \\
(\Gamma_r, v_1 : u = h \mid C_r \mid e) & \text{ when runtime}(\Gamma_r \mid \langle c \rangle[j] : u) = h
\end{align*}
\]

By assumption, the runtime function \(\text{runtime}(\Gamma_r \mid \langle c \rangle[j] : u)\) establishes a derivation \(\Gamma_r \vdash h : u\). From the **THIN-VAR-DEF** rule, we can infer that \(\Gamma_r, v_1 : u = h \vdash e : \text{enum}_0\).

- Suppose the reduction uses the rule **RED-LETSUB-FRAME**.

\[
\begin{align*}
((\Gamma_r, v_2 : u' = \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle) \text{ as } \Gamma_r | C_r \mid \text{let } v_1 : u = v_2[j] \text{ in } e) & \rightarrow \\
(\Gamma_r, v_1 : u = h \mid c \mid e) & \text{ when runtime}(\Gamma_r \mid \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle[j] : u) = h
\end{align*}
\]

By assumption, the runtime function \(\text{runtime}(\Gamma_r \mid \langle c : \text{frame}(tv[t_1, \ldots, t_m]) \rangle[j] : u)\) establishes a derivation \(\Gamma_r \vdash h : u\). From the **THIN-VAR-DEF** rule, we can infer that \(\Gamma_r, v_1 : u = h \vdash e : \text{enum}_0\).

\(\triangle\)

**SetSubscript** There are several cases for subscript assignment on arrays.

- Suppose the reduction uses the rule **RED-SETSUB-TUPLE**.

\[
\begin{align*}
((\Gamma_r, v : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r | C_r \mid v[j] : u \leftarrow h; e) & \rightarrow \\
(\Gamma_r', v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle \mid C_r \mid e) & \text{ when } j \in \{0 \ldots n-1\}
\end{align*}
\]

The proof of typing must end in the rule **TY-SETSUB-TUPLE**:

\[
\begin{align*}
\Gamma_r \vdash v : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c & \Rightarrow \Gamma_r \vdash h : u \\
\Gamma_r \vdash e : \text{enum}_0
\end{align*}
\]

From the premise \(v : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c\), we infer that \(u'\) is the array type \(u \text{ array}\).

From the premise \(h : u\), we know the context \(\Gamma_r, v : u \text{ array} = \langle [h_i]_0^{i-1}, h, [h_i]_{i+1}^{n-1} \rangle\) is well-formed. From the premise \(e : \text{enum}_0\) we conclude \(\Gamma_r, v : u = \langle [h_i]_0^{i-1}, h, [h_i]_{i+1}^{n-1} \rangle \vdash e : \text{enum}_0\).

- Suppose the reduction uses the rule **RED-SETSUB-ARRAY**.

\[
\begin{align*}
((\Gamma_r, v : u' = \langle h_0, \ldots, h_{n-1} \rangle) \text{ as } \Gamma_r | C_r \mid v[j] : u \leftarrow h; e) & \rightarrow \\
(\Gamma_r', v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle \mid C_r \mid e) & \text{ when } j \in \{0 \ldots n-1\}
\end{align*}
\]

The proof of typing must end in the rule **TY-SETSUB-ARRAY**:

\[
\begin{align*}
\Gamma_r \vdash v : u \text{ array} & \Rightarrow \Gamma_r \vdash j : \text{offset} \\
\Gamma_r \vdash h : u & \Rightarrow \Gamma_r \vdash e : \text{enum}_0
\end{align*}
\]

From the premise \(v : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c\), we can infer that \(u'\) is the tuple type \(\langle u_0, \ldots, u_{i-1}, u, u_{i+1}, \ldots, u_{n-1} \rangle_c\). From the premise \(h : u\), we know the context \(\Gamma_r, v : u = \langle h_0, \ldots, h_{i-1}, h, h_{i+1}, \ldots, h_{n-1} \rangle\) is well-formed. Finally, from the premise \(e : \text{enum}_0\) we conclude that \(\Gamma_r, v : u = \langle h_0, \ldots, h_{i-1}, h, h_{i+1}, \ldots, h_{n-1} \rangle \vdash e : \text{enum}_0\).
• Suppose the reduction uses the rule \texttt{RED-SET-UNION}.

\[
\begin{align*}
(G_r, v: u' = tv_j(h_0, \ldots, h_{n-1}) | C_r \mid v[k]: u \leftarrow h \mid e) \rightarrow \\
(G_r, v: u' = tv_j(h_0, \ldots, h_{k-1}, h, h_{k+1}, \ldots, h_{n-1}) | C_r \mid e)
\end{align*}
\]

The proof of typing must end in the rule \texttt{TY-SET-UNION}:

\[
\begin{align*}
\Gamma_r \vdash v : \text{union}(tv[t_1', \ldots, t_m'], \{j\}) \\
\Gamma_r \vdash tv[t_1', \ldots, t_m'] = \left(\left[\tilde{u}_i\right]_0^{n-1} + \left[\tilde{u}_i\right]_0^{k-1}, u, \left[\tilde{u}_i\right]_0^{k+1}, u \right) : \omega_{\text{union}[n]} \\
\Gamma_r \vdash h : u \\
\Gamma_r \vdash e : \text{enum}_0
\end{align*}
\]

From the premise \(\Gamma_r \vdash v : \text{union}(tv[t_1', \ldots, t_m'], \{j\})\) we infer that \(u' = \text{union}(tv[t_1', \ldots, t_m'], \{j\})\).
From the premise \(h : u\), we know the context \(\Gamma_r, v : u = tv_j(h_0, \ldots, h_{n-1})\) is well-formed. Finally, from the premise \(e : \text{enum}_0\) we conclude that \(\Gamma_r, v : u = tv_j(h_0, \ldots, h_{n-1}) \vdash e : \text{enum}_0\).

• Suppose the reduction uses the rule \texttt{RED-SET-RAW-DATA}.

\[
\begin{align*}
((G'_r, v: u' = \langle c \rangle) \text{ as } \Gamma_r \mid C_r \mid v[j]: u \leftarrow h \mid e) \rightarrow \\
(G'_r, v: u' = \langle c' \rangle | C_r \mid e) \\
\text{when runtime}(\Gamma_r | \langle c \rangle[j]: u \leftarrow h) = \langle c' \rangle
\end{align*}
\]

The corresponding type rule is \texttt{TY-SET-RAW-DATA}.

\[
\begin{align*}
\Gamma_r \vdash v : \text{data} \\
\Gamma_r \vdash j : \text{off}tset \\
\Gamma_r \vdash h : u \\
\Gamma_r \vdash e : \text{enum}_0
\end{align*}
\]

We can infer that \(u' = \text{data}\). Since \(\langle c' \rangle : \text{data}\), we conclude that \(\Gamma_r, v : \text{data} = \langle c' \rangle \vdash e : \text{enum}_0\).

• Suppose the reduction uses the rule \texttt{RED-SET-FRAME}.

\[
\begin{align*}
((G'_r, v: u' = \langle c : \text{frame}(tv[t_1, \ldots, t_m])\rangle) \text{ as } \Gamma_r \mid C_r \mid v[j]: u \leftarrow h \mid e) \rightarrow \\
(G'_r, v: u' = \langle c' : \text{frame}(tv[t_1, \ldots, t_m])\rangle | C_r \mid e) \\
\text{when runtime}(\Gamma_r | \langle c : \text{frame}(tv[t_1, \ldots, t_m])\rangle[j]: u \leftarrow h) = \langle c' : \text{frame}(tv[t_1, \ldots, t_m])\rangle
\end{align*}
\]

The corresponding type rule is \texttt{TY-SET-FRAME}.

\[
\begin{align*}
\Gamma_r \vdash v : \text{frame}(tv[t_1, \ldots, t_m]) \\
\Gamma_r \vdash tv[t_1, \ldots, t_m] = \{t^l : \{t^r : u; \ldots\} : \omega_{\text{frame}} \\
\Gamma_r \vdash h : u \\
\Gamma_r \vdash e : \text{enum}_0
\end{align*}
\]

\[
\Gamma_r \vdash v[l\text{label}(tv, t^l, t^r, r_{off})]: u \leftarrow h \mid e : \text{enum}_0
\]

We can infer that \(u' = \text{frame}(tv[[t_i]_1^{m}])\). Since \(\langle c' : \text{frame}(tv[[t_i]_1^{m}])\rangle : \text{frame}(tv[t_1, \ldots, t_m])\), we conclude that \(\Gamma_r, v : u' = \langle c' : \text{frame}(tv[t_1, \ldots, t_m])\rangle \vdash e : \text{enum}_0\).

\(\triangle\)
LetGlobal Suppose that \( e_r = \text{let } v: u = \text{global} \text{ in } e \). The operational rule is Red-LETGLOBAL.

\[
((\Gamma_r', l: u = h) \text{ as } \Gamma_r | C_r | \text{let } v: u = \text{global} \text{ in } e) \rightarrow (\Gamma_r, v: u = h | C_r | e)
\]

The proof of typing must end in the rule Ty-LETGLOBAL.

\[
\begin{array}{c}
\Gamma_r \vdash l : u \\
\Gamma_r, v : u \vdash e : \text{enum}_0
\end{array}
\]

\[
\Gamma_r \vdash \text{let } v: u = \text{global} \text{ in } e : \text{enum}_0
\]

From the premise \( l : u \), we know that \( u' = u \). From the premise \( \Gamma_r, v : u \vdash e : \text{enum}_0 \), we conclude that \( \Gamma_r, v : u = h \vdash e : \text{enum}_0 \). \( \triangle \)

SetGlobal Suppose that \( e_r = \text{global} : u \leftarrow h; e \). The operational rule is Red-SETGLOBAL.

\[
((\Gamma_r', l: u = h') \text{ as } \Gamma_r | C_r | \text{global} : u \leftarrow h; e) \rightarrow (\Gamma'_r, l: u = h | C_r | e)
\]

The proof of typing must end in the rule Ty-SETGLOBAL.

\[
\begin{array}{c}
\Gamma_r \vdash l : u \\
\Gamma_r \vdash a : u \\
\Gamma_r \vdash e : \text{enum}_0
\end{array}
\]

\[
\Gamma_r \vdash \text{global} : u \leftarrow a; e : \text{enum}_0
\]

From the premise \( l : u \), we know that \( u' = u \). From the premise \( \Gamma_r \vdash e : \text{enum}_0 \), we conclude that \( \Gamma'_r, l: u = h \vdash e : \text{enum}_0 \). \( \triangle \)

\[\square\]

7.2 Progress

Theorem 7.2.1 Progress If \( (\Gamma_r | C_r | e_r) \) is a program, and \( e_r \) is not a value \( h \), then there is a program \( (\Gamma_c | C_c | e_c) \) such that \( (\Gamma_r | C_r | e_r) \rightarrow (\Gamma_c | C_c | e_c) \), or \( (\Gamma_r | C_r | e_r) \rightarrow \text{error} \).

We prove this by induction on the length of the proof \( \Gamma_r \vdash e_r : t \).

Type judgments in general require value declarations of the form \( v: t \) rather than value definitions of the form \( v: t = h \). We use the Proof Induction Lemma 6.2.1 throughout this proof to infer use of specific type rules in a partially thinned context \( \Gamma_r \) where some number of thinning rules have been applied.

Variables Suppose \( e_r = E[v] \) and \( \Gamma_r = (\Gamma'_r, v: t' = h) \). The rule Red-ATOM-VAR applies.

\[
(\Gamma_r, v: t = h | C_r | E[v]) \rightarrow (\Gamma_r, v: t = h | C_r | E[h])
\]

In the remainder of the proof, we will consider only the cases for \( E[v] \) where \( \Gamma_r \neq (\Gamma'_r, v: t' = h) \). \( \triangle \)
**Unary operations** Suppose $e_r = E[\text{unop } h]$. By the Expression Typing Lemma 6.4.4, the atom $\text{unop } h$ must be well-typed for some type $t_a$. The only rule that applies is $\text{TY-ATOM\text{-}UNOP}$:

$$
\frac{\Gamma_r \vdash a : \text{arg}(\text{unop}) \quad \Gamma_r \vdash \text{res}(\text{unop}) : \Omega}{\Gamma_r \vdash \text{unop } a : \text{res}(\text{unop})}
$$

According to the unary operator syntax table in Figure 2.5, there are two general cases for $\text{arg}(\text{unop})$: it is either a scalar type, or the unary operator is a type coercion.

- In the type coercion case, the unary reduction $\text{RED\text{-}ATOM\text{-}UNOP}$ always applies.
- To illustrate the scalar case, suppose $\text{arg}(\text{unop}) = \mathbb{Z}_{31}$. From the Numeric Values Lemma 6.7.2, we can assume that the value $h$ is either a constant or a variable. If it is a constant, then the reduction $\text{RED\text{-}ATOM\text{-}UNOP}$ applies. Otherwise, if $h$ is a variable $v$, then it must be defined in the context $\Gamma_r$ with a definition $v : \mathbb{Z}_{31} = b$. Again, from the Numeric Values Lemma 6.7.2, we can infer that $b$ is a constant or a variable (and so it is a heap value). The $\text{RED\text{-}ATOM\text{-}VAR}$ reduction applies.

$\triangle$

**Binary operations** Binary operations are similar to the unary case. Suppose $e_r = E[h_1 \text{ binop } h_2]$. By the Expression Typing Lemma 6.4.4, the atom $h_1 \text{ binop } h_2$ must be well-typed for some type $t_a$. The only rule that applies is $\text{TY-ATOM\text{-}BINOP}$:

$$
\frac{\Gamma_r \vdash a_1 : \text{arg}_1(\text{binop}) \quad \Gamma_r \vdash a_2 : \text{arg}_2(\text{binop}) \quad \Gamma_r \vdash \text{res}(\text{binop}) : \Omega}{\Gamma_r \vdash a_1 \text{ binop } a_2 : \text{res}(\text{binop})}
$$

According to the binary operator syntax table in Figure 2.7, there are two general cases for $\text{arg}_j(\text{binop})$: it is either a scalar type, or the binary operator is a comparison. In the comparison case, the binary reduction $\text{RED\text{-}ATOM\text{-}BINOP}$ always applies. Otherwise, according to the Numeric Values Lemma 6.7.2, the atoms $h_1$ and $h_2$ must be either constants or variables. One of the reductions $\text{RED\text{-}ATOM\text{-}BINOP}$ or $\text{RED\text{-}ATOM\text{-}VAR}$ applies.

$\triangle$

**Type application** Suppose $e_r = E[\text{ty\text{-}apply}[u][h_f, u_1, \ldots, u_n]]$. By Expression Typing Lemma 6.4.4, the atom $\text{ty\text{-}apply}[u][h_f, u_1, \ldots, u_n]$ must be well-typed for some type $t_a$. The only rule that applies is the $\text{TY\text{-}ATOM\text{-}TY\text{-}APPLY}$ rule:

$$
\frac{\Gamma_r \vdash [u_i : \omega]_m \quad \Gamma_r \vdash h_f : \forall \alpha_1, \ldots, \alpha_m. t \quad \Gamma_r \vdash u = t[u_i/\alpha_i]_m : \omega}{\Gamma_r \vdash \text{ty\text{-}apply}[u][h_f, u_1, \ldots, u_n] : t[u_i/\alpha_i]_m}
$$

The only rule that can be used to prove the premise $\Gamma_r \vdash h_f : \forall \alpha_1, \ldots, \alpha_n. t$ is the $\text{TY\text{-}ATOM\text{-}VAR}$ rule:

$$
\frac{\Gamma_r, v : \forall \alpha_1, \ldots, \alpha_n. t \vdash \phi}{\Gamma_r, v : \forall \alpha_1, \ldots, \alpha_n. t \vdash v : \forall \alpha_1, \ldots, \alpha_n. t}
$$

That is, the value $h_f$ must be a variable $v$, and context $\Gamma_r$ must include the declaration $v : \forall \alpha_1, \ldots, \alpha_n. t$. Since the context is fully-defined and well-formed, it contains a definition $v : \forall \alpha_1, \ldots, \alpha_n. t = b$, which is well-formed using the judgment $\text{THIN\text{-}VAR\text{-}DEF}$.

90
\[ \hat{\Gamma}_r. v; \forall \alpha_1, \ldots, \alpha_n. t \vdash b : \forall \alpha_1, \ldots, \alpha_n. t \quad \hat{\Gamma}_r. v; \forall \alpha_1, \ldots, \alpha_n. t \vdash \emptyset \]

\[ \hat{\Gamma}_r. v; \forall \alpha_1, \ldots, \alpha_n. t = v \vdash \emptyset \]

The only possible value \( b \) for \( v \) is specified by the **TY-STOREFun-POLY** rule.

\[ \hat{\Gamma}_r, [\alpha_i : \omega]_1^m \vdash \lambda v_1, \ldots, v_n. e : (u_1, \ldots, u_n) \rightarrow t \]

\[ \hat{\Gamma}_r \vdash \Lambda \alpha_1, \ldots, \alpha_m. \lambda v_1, \ldots, v_n. e : \forall \alpha_1, \ldots, \alpha_m. (u_1, \ldots, u_n) \rightarrow t \]

That is, \( b \) must be a function \( \Lambda \alpha_1, \ldots, \alpha_m. \lambda v_1, \ldots, v_n. e. \)

This means that the context \( \Gamma_t \) has the form

\[ \Gamma_t = \Gamma_r', f : \forall \alpha_1, \ldots, \alpha_m. (u_1, \ldots, u_n) \rightarrow t = \Lambda \alpha_1, \ldots, \alpha_m. \lambda v_1, \ldots, v_n. e. \]

The operational rule **RED-ATOM-APPLY** applies.

\[ \Gamma_t, [u' = \Lambda \alpha_1, \ldots, \alpha_m. b] \text{ as } \Gamma_r \mid C_r \mid E[\text{ty} \_ \text{apply}[u][f, u_1, \ldots, u_m]] \rightarrow \]

\[ \Gamma_t, [\alpha_i : \omega = u_i]_1^m, g : u = b \mid C_r \mid E[g] \]

**Type abstraction** Suppose \( e_r = E[\text{ty} \_ \text{pack}[u][v_1, u_1, \ldots, u_{n-1}]]. \) By **expression typing** Lemma 6.4.4, the atom \( \text{ty} \_ \text{pack}[u][v_1, u_1, \ldots, u_{n-1}] \) must be well-typed for some type \( t_a \). The only rule that applies is the **TY-ATOMTyPack** rule.

\[ \hat{\Gamma}_r \vdash [u_i : \omega]_1^m \quad \hat{\Gamma}_r \vdash v_1 : t[[u_i / \alpha_i]_1^m] \quad \hat{\Gamma}_r \vdash u = \exists \alpha_1, \ldots, \alpha_m. t : \omega \]

\[ \hat{\Gamma}_r \vdash \text{ty} \_ \text{pack}[u](v_1, u_1, \ldots, u_m) : \exists \alpha_1, \ldots, \alpha_m. t \]

The only rule that can be used to prove the premise \( \hat{\Gamma}_r \vdash v_1 : t[[u_i / \alpha_i]_1^m] \) is the **TY-ATOMVAR** rule.

\[ \hat{\Gamma}_r, v : t[[u_i / \alpha_i]_1^m] \vdash \emptyset \]

\[ \hat{\Gamma}_r, v : t[[u_i / \alpha_i]_1^m] \vdash v : t[[u_i / \alpha_i]_1^m] \]

That is, atom \( a \) must be a variable, and context \( \Gamma_t \) must include the declaration \( v : t[[u_i / \alpha_i]_1^m] \). Since the context is fully-defined and well-formed, it contains a definition \( v : t[[u_i / \alpha_i]_1^m] = b \). The operational rule **RED-ATOM-Pack** applies.

\[ ((\Gamma_r, v_1 : u' = b) \text{ as } \Gamma_r \mid C_r \mid E[\text{ty} \_ \text{pack}[u][v_1, u_0, \ldots, u_{n-1}]] \rightarrow \]

\[ (\Gamma_r, v_2 : u = \text{ty} \_ \text{pack}[u](v_1, u_0, \ldots, u_{n-1}) \mid C_r \mid E[v_2]) \]

**Abstraction unpacking** Suppose \( e_r = E[\text{ty} \_ \text{unpack}(v_1)]. \) By the **expression typing** Lemma 6.4.4, the atom \( \text{ty} \_ \text{unpack}(v_1) \) must be well-typed for some type \( t_a \). The only rule that can be applied is **TY-ATOMTyUnpack**.
\[ \Gamma_r \vdash v : \exists \alpha_0, \ldots, \alpha_{m-1}. t \]
\[ \Gamma_r \vdash \text{ty\_unpack}(v) : t[[v, i/\alpha_i]_0^{m-1}] \]

The only rule that can be used to prove the premise \( \Gamma_r \vdash v : \exists \alpha_1, \ldots, \alpha_m. t \) is the \textbf{Ty-AtomVAR} rule.

\[ \Gamma_r, v : \exists \alpha_1, \ldots, \alpha_m. t \vdash \phi \]
\[ \Gamma_r, v : \exists \alpha_1, \ldots, \alpha_m. t \vdash v : \exists \alpha_1, \ldots, \alpha_m. t \]

That is, the context \( \Gamma_r \) must include the declaration \( v : \exists \alpha_1, \ldots, \alpha_m. t \). Since the context is fully-defined and well-formed, it contains a definition \( v : \exists \alpha_1, \ldots, \alpha_m. t = b \), which is well-formed using the judgment \textbf{Thin-Var-Def}.

\[ \Gamma_r, v : \exists \alpha_1, \ldots, \alpha_m. t \vdash b : \exists \alpha_1, \ldots, \alpha_m. t \]
\[ \Gamma_r, v : \exists \alpha_1, \ldots, \alpha_m. t \vdash \phi \]
\[ \Gamma_r, v : \exists \alpha_1, \ldots, \alpha_m. t = b \vdash \phi \]

The only possible value \( b \) for \( v \) is specified by the \textbf{Ty-AtomTyPack} rule.

\[ \Gamma_r \vdash [u_i : \omega]_1^m \]
\[ \Gamma_r \vdash v : t[[u_i/\alpha_i]_0^{m-1}] \]
\[ \Gamma_r \vdash u = \exists \alpha_1, \ldots, \alpha_m. t : \omega \]
\[ \Gamma_r \vdash \text{ty\_pack}[u](v, u_1, \ldots, u_m) : \exists \alpha_1, \ldots, \alpha_m. t \]

That is, \( b \) must be a packed expression \( \text{ty\_pack}[u](v, u_1, \ldots, u_m) \). This means that the context \( \Gamma_r \) has the form \( \Gamma'_r, v : \exists \alpha_1, \ldots, \alpha_m. t = \text{ty\_pack}[u](v, u_1, \ldots, u_m) \). The operational rule \textbf{Red-Atom-Unpack} applies.

\[ \left( \Gamma'_r, v_1 \vdash u = \text{ty\_pack}[\exists \alpha_i [0]^{m-1}. t](v, [u_i]_0^{m-1}) \right) \rightarrow \left( \Gamma_r, [\alpha_i : \omega = u_i]_0^{m-1}, v_2 : t = v \vdash E[v_2] \right) \]

For the remaining cases, we can assume that the expression \( e_r \) cannot be expressed as a context \( E[a] \). That is, the atoms in the expression have been fully evaluated.

\textbf{LetAtom} Suppose \( e_r = \text{let} \; v : t = h \; \text{in} \; e \). The reduction \textbf{Red-LetAtom} applies.

\[ (\Gamma_r \mid C_r \mid \text{let} \; v : t = h \; \text{in} \; e) \rightarrow (\Gamma_r, v : u = h \mid C_r \mid e) \]

\[ \triangle \]

\textbf{TailCall} Suppose \( e_r = h(h_1, \ldots, h_n) \). The type judgment must use the \textbf{Ty-TailCall} rule.

\[ \Gamma_r \vdash h : \text{enum}_0 \]
\[ \Gamma_r \vdash [h_i : u_i]_1^n \]
\[ \Gamma_r \vdash h(h_1, \ldots, h_n) : \text{enum}_0 \]
By the Function Values Lemma 6.7.3, the value \( h \) must be a variable \( v \), and the definition of \( v \) must have the form \( \lambda v_1, \ldots, v_n. e \). The Red-TailCall rule applies.

\[
((\Gamma_r, v; (u_1, \ldots, u_n) \rightarrow t = \lambda v_1, \ldots, v_n. e) \text{ as } \Gamma_r | C_r | v(h_1, \ldots, h_n)) \rightarrow
(\Gamma_r, [v_i: u_i = h_i]_r^n | C_r | e)
\]

\( \triangle \)

Special Calls Suppose \( e_r = \text{special} S \) is a special call. There are several cases.

- Suppose \( e_r = \text{migrate } [j, h_p, h_o] h_{\text{fun}}(h_1, \ldots, h_n) \). The typing rule is \( \text{TY-SysMigrate} \).

\[
\begin{align*}
\Gamma_r & \vdash h_{\text{ptr}} : \text{data} \\
\Gamma_r & \vdash h_{\text{off}} : \mathbb{Z}_{32}^{\text{signed}} \\
\Gamma_r & \vdash [h_i : t_i]^n \\
\Gamma_r & \vdash h_f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0 \\
\Gamma_r & \vdash \text{migrate } [j, h_{\text{ptr}}, h_{\text{off}}] h_f(h_1, \ldots, h_n) : \text{special enum}_0
\end{align*}
\]

By the Function Values Lemma 6.7.3, since \( h_f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0 \), it must be defined in the context as a function \( \lambda v_1, \ldots, v_n. e \). The Red-SysMigrate rule applies.

\[
(\Gamma_r | C_r | \text{special migrate } [j, h_{\text{ptr}}, h_{\text{off}}] f(h_1, \ldots, h_n)) \rightarrow (\Gamma_r | C_r | f(h_1, \ldots, h_n))
\]

- Suppose \( e_r = \text{atomic } h_f(h_{\text{const}}, h_1, \ldots, h_n) \). The corresponding typing rule is \( \text{TY-Atomic} \).

\[
\begin{align*}
\Gamma_r & \vdash h_{\text{const}} : \mathbb{Z}_{32}^{\text{signed}} \\
\Gamma_r & \vdash [h_i : t_i]^n \\
\Gamma_r & \vdash h_f : (\mathbb{Z}_{32}^{\text{signed}}, t_1, \ldots, t_n) \rightarrow \text{enum}_0 \\
\Gamma_r & \vdash \text{atomic } h_f(h_{\text{const}}, h_1, \ldots, h_n) : \text{special enum}_0
\end{align*}
\]

By the Function Values Lemma 6.7.3, since \( h_f : (h_{\text{const}}, h_1, \ldots, h_n) \rightarrow \text{enum}_0 \), the value \( h_f \) must be a variable \( f \). The Red-Atomic rule applies.

\[
(\Gamma_r | C_r | \text{special atomic } f(h_{\text{const}}, h_1, \ldots, h_n)) \rightarrow (\Gamma_r | C_r | f(h_{\text{const}}, h_1, \ldots, h_n))
\]

- Suppose \( e_r = \text{rollback } [h_{\text{level}}, h_{\text{const}}] \). The corresponding typing rule is \( \text{TY-AtomicRollback} \).

\[
\begin{align*}
\Gamma_r & \vdash h_{\text{level}} : \mathbb{Z}_{32}^{\text{signed}} \\
\Gamma_r & \vdash h_{\text{const}} : \mathbb{Z}_{32}^{\text{signed}} \\
\Gamma_r & \vdash \text{rollback } [h_{\text{level}}, h_{\text{const}}] : \text{special enum}_0
\end{align*}
\]

By the Numeric Values Lemma 6.7.2, the value \( h_{\text{level}} \) must be a variable or a constant. In the former case, the Red-Atom-Var rule applies. Otherwise, let \( i = h_{\text{level}} \), and suppose the checkpoint context \( C_r \) contains \( m \) checkpoints \( C_m, \ldots, C_1 \).

If \( 1 \leq i \leq m \), then the Red-AtomicRollback-1 rule applies.

\[
(\Gamma_r | C_m; \ldots; C_i = (\Gamma_r, f(o, h_1, \ldots, h_n)); \ldots; C_1 | \text{special rollback } [i, j]) \rightarrow
(\Gamma_r | C_i; C_{i-1}; \ldots; C_1 | f(j, h_1, \ldots, h_n))
\]

when \( i \in \{1 \ldots m\} \)

If \( i = 0 \land m > 0 \), then the Red-AtomicRollback-2 rule applies.

\[
(\Gamma_r | C_m = (\Gamma_r, f(o, h_1, \ldots, h_n)); \ldots; C_1 | \text{special rollback } [0, j]) \rightarrow
(\Gamma_r | C_m; \ldots; C_1 | f(j, h_1, \ldots, h_n))
\]
If \( i < 0 \lor i > m \), then the Red-Atomic-Rollback-Error rule applies.

\[
\begin{align*}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \text{special rollback } [i, j]) & \rightarrow \text{error} \\
\text{when } i \notin \{0, \ldots m\} \lor m = 0
\end{align*}
\]

- Suppose \( e_r = \text{commit } [h_{\text{level}}] \, h_f(h_1, \ldots, h_n) \). The typing rule is Ty-AtomicCommit.

\[
\begin{align*}
\widehat{\Gamma}_r : h_{\text{level}} : \mathbb{Z}_{32} & \\
\Gamma_r \vdash h_{\text{level}} \colon \mathbb{Z}_{32} & \\
\widehat{\Gamma}_r : [h : t_i]_1^n & \\
\Gamma_r \vdash h_f : (t_1, \ldots, t_n) \rightarrow \text{enum}_0 & \\
\widehat{\Gamma}_r \vdash \text{commit } [h_{\text{level}}] \, h_f(h_1, \ldots, h_n) : \text{specialenum}_0
\end{align*}
\]

By the Numeric Values Lemma 6.7.2, the value \( h_{\text{level}} \) must be a variable or a constant. In the former case, the Red-Atom-Var rule applies. Otherwise, let \( i = h_{\text{level}} \), and suppose the checkpoint context \( C_r \) contains \( m \) checkpoints \( C_m, \ldots, C_1 \).

If \( 1 \leq i \leq m \), then the Red-Atomic-Commit-1 rule applies.

\[
\begin{align*}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \text{special commit } [i] \, f(h_1, \ldots, h_n)) & \rightarrow \\
(\Gamma_r \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) & \\
\text{when } i \in \{1, \ldots m\}
\end{align*}
\]

If \( i = 0 \land m > 0 \), then the Red-Atomic-Commit-2 rule applies.

\[
\begin{align*}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \text{special commit } [0] \, f(h_1, \ldots, h_n)) & \rightarrow \\
(\Gamma_r \mid C_{m-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n))
\end{align*}
\]

If \( i < 0 \lor i > m \), then the Red-Atomic-Commit-Error rule applies.

\[
\begin{align*}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \text{special commit } [i] \, f(h_1, \ldots, h_n)) & \rightarrow \text{error} \\
\text{when } i \notin \{0, \ldots m\} \lor m = 0
\end{align*}
\]

\[\triangle\]

**Match** Suppose \( e_r = \text{match } h \text{ with } [s_j \mapsto e_j]_{j=1}^n \). The type judgment must use one of the match rules. We illustrate with the rule Ty-Match-Int.

\[
\begin{align*}
[s_i \in \text{set } t]_1^n & \\
\bigcup_{i=1}^n s_i = \mathbb{Z}_{31} & \\
\widehat{\Gamma}_r : a : \mathbb{Z}_{31} & \\
\widehat{\Gamma}_r : [e_i : t]_1^n & \\
\widehat{\Gamma}_r : \text{match } a \text{ with } [s_i \mapsto e_i]_1^n : t
\end{align*}
\]

Since \( a \) has scalar type \( \mathbb{Z}_{31} \), by the Numeric Values Lemma 6.7.2 it must be either a variable or a constant. In the former case, the Red-Atom-Var rule applies.

In the latter case, the type judgment also requires that the sets \( s \) be total, that is \( \bigcup_{i=1}^n s_j = \mathbb{Z}_{31} \). Since \( h \) must belong to one of the sets \( s_k \), the operation rule Red-Match-Int applies.

\[
\begin{align*}
(\Gamma_r \mid C_r \mid \text{match } i \text{ with } [s_j \mapsto e_j]_{j=1}^n) & \rightarrow (\Gamma_r \mid C_r \mid e_k) \\
\text{when } (i \in s_k) \land \forall j \in \{1, \ldots, k-1\}. (i \notin s_j)
\end{align*}
\]

\[\triangle\]

**Allocation** Suppose \( e_r = \text{let } v = \text{alloc in } e \). One of the allocation reductions Red-Alloc-Tuple, Red-Alloc-Union, or Red-Alloc-Array always applies.

\[\triangle\]
**Subscripting** Suppose $e_r = \text{let } v; t = h_1[h_2] \text{ in } e$. The type judgment requires application of one of the subscripting type rules. We illustrate with $\text{TY-LETSUB-TUPLE}$.

$$
\Gamma_r \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \Gamma_r, v_1 : u \vdash e : t
$$

$$
\Gamma_r \vdash \text{let } v_1 : u = a_1[\text{offset}_c^{\text{const}}(j)] \text{ in } e : t
$$

In this case, $h_1$ must be a variable $v_1$ with type $\langle u_0, \ldots, u_{j-1}, t_1, u_{j+1}, \ldots, u_{n-1} \rangle_c$, and $h_2$ must be a constant $\text{offset}_c^{\text{const}}(i)$.

Since $v_1$ has a tuple type, its value is determined by the $\text{TY-STORETUPLE}$ rule.

$$
\Gamma_r \vdash \langle a_1, \ldots, a_n \rangle : \langle t_1, \ldots, t_n \rangle_{\text{tuple}_c^{\text{class}}} \quad \Omega
$$

That is $v_1$ must be a tuple $v_1 : \langle u_0, \ldots, u_{j-1}, t_1, u_{j+1}, \ldots, u_{n-1} \rangle_c = \langle a_1, \ldots, a_n \rangle$.

The reduction $\text{RED-LETSUB-TUPLE}$ applies.

$$
\Gamma_r, v_1 : u = h_j \vdash e \quad \text{as} \quad \Gamma_r \mid C_r \mid \text{let } v_1 : u = v_2[j] \text{ in } e \\
\Gamma_r \vdash \text{let } v_1 : u = h_j \mid C_r \mid e
$$

\[\triangle\]

**Assignment** Suppose $e_r = h_1[h_2] : t \leftarrow h_3 ; e$. The type judgment requires application of one of the subscript assignment type rules. We illustrate with $\text{TY-SETSUB-TUPLE}$.

$$
\Gamma_r \vdash h_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \Gamma_r \vdash a_3 : u \quad \Gamma_r \vdash e : t
$$

$$
\Gamma_r \vdash h_1[\text{offset}_c^{\text{const}}(j)] : u \leftarrow a_3 ; e : t
$$

In this case, $h_1$ must be a variable $v_1$ with type $\langle u_0, \ldots, u_{j-1}, t_1, u_{j+1}, \ldots, u_{n-1} \rangle_c$, and $h_2$ must be a constant $\text{offset}_c^{\text{const}}(i)$.

Since $v_1$ has a tuple type, its value is determined by the $\text{TY-STORETUPLE}$ rule.

$$
\Gamma_r \vdash \langle a_1, \ldots, a_n \rangle : \langle t_1, \ldots, t_n \rangle_{\text{tuple}_c^{\text{class}}} \quad \Omega
$$

That is $v_1$ must be a tuple $v_1 : \langle u_0, \ldots, u_{j-1}, t_1, u_{j+1}, \ldots, u_{n-1} \rangle_c = \langle a_1, \ldots, a_n \rangle$.

The reduction $\text{RED-SETSUB-TUPLE}$ applies.

$$
\Gamma_r, v_1 : u' = \langle h_0, \ldots, h_{n-1} \rangle \mid C_r \mid v[j] : u \leftarrow h ; e \quad \text{as} \quad \Gamma_r \mid C_r \mid v[j] : u \leftarrow h ; e \\
\Gamma_r, v_1 : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle \mid C_r \mid e
$$

\[\triangle\]

**LetGlobal** Suppose $e_r = \text{let } v; t = \text{global} \mid e$. The corresponding typing rule is $\text{TY-LETGLOBAL}$.

$$
\Gamma_r \vdash t : u \quad \Gamma_r, v : u \vdash e : t
$$

$$
\Gamma_r \vdash \text{let } v ; u = \text{global} \mid e : t
$$

95
By the Variable Declarations Lemma 6.7.1, the label \( l \) must be defined as part of the context. The Red-LetGlobal rule applies.

\[
((\Gamma'_r, l: u = h) \text{ as } \Gamma_r \mid \mathcal{C}_r \mid \text{let } v; u = \text{global} l \text{ in } e) \rightarrow (\Gamma_r, v; u = h \mid \mathcal{C}_r \mid e)
\]

\( \triangle \)

**SetGlobal** Suppose \( e_r = \text{global} l: t \leftarrow h; e \). The corresponding typing rule is TY-SetGlobal.

\[
\frac{\hat{\Gamma}_r \vdash l: u \quad \hat{\Gamma}_r \vdash a: u \quad \hat{\Gamma}_r \vdash e: t}{\hat{\Gamma}_r \vdash \text{global} l: u \leftarrow a; e : t}
\]

By the Variable Declarations Lemma 6.7.1, the label \( l \) must be defined as part of the context. The Red-SetGlobal rule applies.

\[
((\Gamma'_r, l: u = h') \text{ as } \Gamma_r \mid \mathcal{C}_r \mid \text{global} l: u \leftarrow h; e) \rightarrow (\Gamma'_r, l: u = h \mid \mathcal{C}_r \mid e)
\]

\( \triangle \)

\( \square \)
Chapter 8

Runtime Implementation

The FIR is machine-independent, and the Mojave compiler architecture is designed to support multiple backends, including both native-code and interpreted runtimes. Currently, our main runtime implementation is a native-code runtime for the Intel IA32 architecture. Object code generation is performed in two stages: the FIR is first translated to a “Machine Intermediate Representation” (MIR), which introduces runtime safety checks in a machine-independent form, and then the final object code is generated for the target architecture from the MIR program. We do not discuss the MIR language in detail here; the language itself is similar to the FIR with a simpler type system, and the process of generating MIR code is a straightforward elaboration of the FIR code.

The runtime implementation manages several tasks, including execution of runtime type-checks for subscript operations, garbage collection, process migration, and atomic transactions. To complicate matters, a faithful C pointer semantics rules out direct use of data relocation (which occurs when a process migrates, or during heap compaction). To address these matters we introduce several auxiliary data structures and invariants.

8.1 Runtime data structures and invariants

The runtime consists of the following parts and invariants.

- A *heap*, containing the data for tuples, arrays, unions, rawdata, and frames. A data value in the heap is called a *block*, and the heap contains multiple (possibly non-contiguous) blocks.
- A *text area*, containing the program code. The text area is immutable at all times except during process migration.
- A set of *registers*. Each variable in the program is assigned to a register. At any time during program execution, a register may contain a value in one of several machine types: a pointer into the heap, a function pointer, or a numerical value. The machine type is statically determined from the variable’s type in the FIR. Register spills have the same properties as registers.

  **Invariant**: if a register contains a pointer, it contains the address of a block in the heap; if a register contains a function pointer, it contains the address of a function entry point in the text area.

- A *pointer table*, containing pointers to all valid data blocks in the heap.
Invariant: all non-empty entries in the pointer table contain pointers to valid blocks in the heap, and every block in the heap has an entry in the pointer table.

- A function table, containing function pointers to all valid higher-order functions. The function table is immutable, except during process migration.

Invariant: all entries in the function table contain the address of a function entry point in the text area.

- A checkpoint record, containing descriptions of all live program checkpoints. Checkpoints are discussed in Section 8.3.

### 8.1.1 Data blocks and the heap

The heap represents the FIR store, and it contains the store values \( b \) defined in Figure 3.2, which we call blocks. The runtime representation of a block contains two parts: a header that describes the size and type of information stored in the block, and a value area containing the contents of the block.

There are two types of data blocks in the heap. Unsafe data corresponds to values of type \textbf{data} and \textbf{frame}(\textit{tv}[t_1, \ldots, t_n]). Safe data is type-safe ML data, and corresponds to the \langle t_1, \ldots, t_n \rangle_{\textit{tuple-class}}, \textit{t array}, and \textbf{union}(\textit{tv}[t_1, \ldots, t_n], \textit{set}) types.

The contents of unsafe data are not explicitly typed in the FIR, and safety checks are required to ensure the data is interpreted properly. Any pointer read from an unsafe block must be checked to ensure it is a valid pointer, and the bounds must be checked any time an unsafe block is dereferenced. In contrast, the contents of safe block data are typed in the FIR, and many safety checks can be omitted\(^1\). The garbage collector can use explicit FIR types to identify pointers embedded in safe block data, but it must use a more conservative algorithm to determine which values are pointers in unsafe block data\(^2\).

The pointer table contains the address of each valid live block in the heap. A block header has three parts: it contains 1) a tag that identifies the block type (unsafe or safe), 2) an index into the pointer table identifying the pointer for this block, and 3) a nonnegative number that indicates the size of the block. The tag field is overloaded to indicate the union case for data in a disjoint union. The header also contains bits used by garbage collection and transactions.

A null pointer is always a valid pointer to a zero-size block. A null pointer is allocated an entry in the pointer table and may be manipulated like any other base pointer, but any attempt to dereference it will result in a runtime exception. Certain source-level languages require a different null pointer for various data types\(^3\); for these languages, a different zero-size block is allocated in advance for each type.

### 8.1.2 Pointer table

The pointer table \textit{ptable} effectively acts as a segment table for the blocks (segments) in the heap. It supports several features, including migration and transactions, but its main purpose is to allow for relocation and safety for C data areas. The pointer table is implemented in software, however its design is compatible with a hardware implementation for increased efficiency.

\(^1\)Safety checks cannot be omitted on data after a successful migration, unless the two machines are mutually trusting. By default, the destination machine generates safety checks on all data unless the original program was compiled with the \texttt{-unsafe} compiler option.

\(^2\)As a consequence, the garbage collector may consider certain blocks to be live beyond their real live range.

\(^3\)For example, in Java there are different null pointers for each object type, which are not considered equivalent.
The pointer table contains an entry pointing to all allocated data blocks. Source-level C pointers are represented in the runtime as (base + offset) pairs. The base pointer always points to the beginning of a data block in the heap, and the offset is a signed byte index relative to the base. Base pointers are never stored directly in the heap. Instead, the base pointer is stored as an index to an entry in the pointer table, which contains the actual address of the beginning of the data block.

![Pointer Table Diagram](image)

Figure 8.1: Pointer table representation

The pointer table serves several purposes. First, it provides a simple mechanism for identifying and validating data pointers in aggregate blocks. When an index $i$ for a base pointer is read from the heap, the following steps are performed:

1. $i$ is checked against the size of the pointer table to verify if it is a valid index.
2. The value $p$ is read from the $i^{th}$ entry in the pointer table.
3. $p$ is checked to ensure it is not free (that it points into the heap).\(^4\)

After these steps, $p$ is always a valid pointer to the beginning of a block. For additional safety, we can verify that the index stored in $p$'s block header matches index $i$. These steps can be performed in a small number of assembly instructions, requiring only two branch points.

The second purpose of the pointer table is to support relocation. If the heap is reorganized by garbage collection or process migration, the pointer table (and registers) are updated with the new locations, but the heap values themselves are preserved. This level of transparency has a cost: in addition to the execution overhead, the header of each block in the heap contains an index. In the IA32 runtime, if we include the pointer table overhead, the overhead is in excess of 12 bytes per block.

### 8.1.3 Function pointers

Function pointers are managed through the function table `fun_table` that serves the same purpose as the pointer table for heap data. A function stub must be generated for each FIR function that escapes (higher-

\(^4\)Empty pointer table entries are always odd values, whereas valid heap pointers are always even values.
order functions). Each function stub has a function header, and the function table contains the addresses of all the escaping-function headers. To ease some of the runtime safety checks, each function stub is formatted with a block header that indicates that the function is a data block with zero-size. As with block pointers, function pointers are represented in the heap as indexes into the function table.

The function header also contains an arity tag, used to describe the types of the arguments. Arity tags are used when a function is called to ensure that a function is called arguments that are compatible with the function’s signature. The arity tags are integer identifiers, computed at link time from the function signatures. The signatures themselves are generated based on the primitive architecture types, not the high-level FIR types. When a function is called, the arguments have the same arity tag as the function signature, or the runtime raises an exception.

Functions may be invoked in a tail-call directly by specifying the function name, or indirectly through a function pointer. Unlike direct function calls, indirect calls through a function pointer use a standardized calling convention. The escape stub is responsible for moving arguments from the standard registers to the registers expected by the original function.

8.1.4 Pointer safety

The runtime operations for load \( \text{runtime}(\Gamma \mid \langle c \rangle [i] : t) \) and store \( \text{runtime}(\Gamma \mid \langle c \rangle [i] : t \leftarrow h) \) are guaranteed to be type-safe, even for unsafe blocks. The runtime safety check for a load operation is performed as follows.

1. The index \( i \) is compared with the bounds of block \( \langle c \rangle \); an exception is raised if the index is out-of-bounds.
2. The value \( h \) at location \( i \) is retrieved, and a safety check is performed.
   - If \( t \) represents a pointer, then \( h \) should be an index into the pointer table. If \( h \) is a valid pointer table index, and the entry \( ptable[h] \) is a valid pointer \( p \), the result of the load is \( p \).
   - If \( t \) represents a function pointer, then \( h \) should be an index into the function table. If \( h \) is a value function table index, the result of the load is \( ftable[h] \).
   - Otherwise, \( h \) does not represent a pointer, and the result of the load is \( h \).

The safety check for a store operation is somewhat simpler. For a store operation \( \text{runtime}(\Gamma \mid \langle c \rangle [i] : t \leftarrow h) \), the runtime invariants guarantee that if \( t \) represents a pointer, then \( h \) is a valid pointer to a block in the heap; and if \( t \) is of function type, then \( h \) is a valid pointer to a function header. In these two cases (after a bounds-check on the index \( i \)) the index for \( h \) is stored. If \( t \) does not represent either kind of pointer, the value \( h \) is stored directly.

8.2 Process migration

To facilitate fault-tolerant computing, we would like to introduce a level of abstraction between the processes that are running in a distributed system, and the specific machines they are running on. Each process should

---

\(^5\) An escaping value is any value which is passed by reference to a function, or whose address is taken. An escaping function is a function whose address is taken and stored in a function pointer.

\(^6\) This must be checked at runtime since C permits function pointers to be coerced arbitrarily.

\(^7\) The primitive architecture types are currently value, pointer addr, pointer block, poly, and function o.
see the distributed cluster as a single machine and a single shared resource pool, rather than a collection of distinct nodes each with their own set of resources. Processes should have a lifetime greater than that of any particular machine in the cluster, so that the process continues running even when one or many machines it is running on fail.

In a distributed system, a process will execute on a specific machine with a particular architecture. Since individual nodes in a cluster may fail at any time, a mechanism for migrating a process from one machine to another is an essential tool for fault-tolerance. Such a mechanism needs to perform three operations: a pack operation to capture the entire state of the process, including the program counter, all register values, heap data, and code; a transmit operation to transmit the state of the process to a target machine; and an unpack operation to reconstruct the process state on the target machine and resume execution. Collectively, this sequence of operations is referred to as process migration.

Process migration should be architecture-independent, to allow for distributed clusters of heterogeneous nodes. Also, process migration should be safe; the remote machine receiving the program should be able to verify that the program type-checks and that heap values are used in a proper manner. If the remote machine can verify that a received program is safe, then we can use process migration in environments where machines in the cluster do not trust each other entirely, such as the wide-area computing clusters on the Internet.

Note that since process migration requires pack and unpack operations, it is fairly straightforward to extend the mechanism to support saving the process state to a file for later execution, and to write checkpoint files while the process is running that contain snapshots of the full process state. In the event of a later failure, the process can be recovered from this file using the unpack operation.

### 8.2.1 Using process migration in the FIR

In the FIR, migration is expressed using the special-call mechanism. Special-calls are similar to tail-calls, taking a function name and argument list, but a special-call implies some special action is performed before the function is invoked. Since many FIR transformations are not concerned with the special action, these transformations may treat a special-call exactly as a tail-call, simplifying their implementation.

Process migration is expressed in the FIR using the special-call migrate \([i, a_{\text{ptr}}, a_{\text{off}}] \ f(a_1, \ldots, a_n)\). The first three arguments indicate how the migration should be performed, and are not passed as arguments to \(f\). The argument \(i\) is an integer, unique among all migration calls in the program. It is used by the backend to determine where program execution resumes after a successful migration. The pointer \((a_{\text{ptr}}, a_{\text{off}})\) refers to a null-terminated string that determines the migration target. The string includes information on what protocol to use.

There are three protocols that may be used for process migration. The first is the migrate protocol, which will actually send the entire state of a process to another machine, and terminate the process on the original machine. If migration fails for any reason, the process will continue to execute on the original machine. The process has no way of directly determining what machine it is running on, or observing a successful migration\(^8\); this encourages an abstraction between the process and the machine it is running on. The fact that the process does not observe the result of migration is reflected in the operational semantics.

In order to migrate to another machine, the remote machine must run the migration server, \texttt{mcc-migrated}. This is a version of the compiler which will listen for incoming migration requests, recompile any inbound

\(^8\)In future implementations, we may provide an external function which reports on the status of a prior migration attempt. Also, a process may indirectly observe the result of a migration since we permit direct linking to \texttt{libc} functions.
processes on the new machine, and reconstruct their state before executing them.

The other two protocols write the process state to a file for later execution. The suspend protocol writes the process state to a file, and terminates the process if it is successfully written. In contrast, the checkpoint protocol will continue running the process even when the file is successfully written. The latter protocol is useful for taking snapshots of a process state, as a crude rollback mechanism or in anticipation of possible machine failure in the near future. The mcc-resurrect program is used to resume process execution from a state file.

8.2.2 Runtime support for migration

Process migration requires two key features from the runtime. First, to support the pack operation, the runtime must be able to collect the entire process state; for unpack, it must be able to restore the process state from a previous pack operation. Second, the migration should be architecture-independent.

The implementation of the pack and unpack operations is relatively straightforward. Since all heap data and function pointers in the heap are represented indirectly as indices, the heap data is not modified by a migration, even if the data are relocated. The pack operation first performs garbage collection and then packs the live data in the heap, the pointer table, the program text, and the registers into a message that can be stored or transmitted.

In order to achieve architecture independence, we never migrate the actual executable text. Instead we migrate the FIR code for the program. The location index \( i \) in the migration call is used to correlate the runtime execution point with a corresponding execution point in the FIR. On an unpack operation, the FIR code is type-checked, recompiled, and execution is resumed. Note that for C programs, the size and byte-ordering of the various data types must conform to a uniform standard.

It is possible to migrate values stored in hardware registers across architectures. Note that the only live variables across migration are the arguments \( a_1, \ldots, a_n \) passed to function \( f \). This corresponds exactly to the set of register values which will be live during migration. To migrate these values, the backend packs them into a newly allocated block in the heap, taking care to convert any real pointers into index values. This allows us to use an architecture-dependent representation of values in the registers, and also provides safety checks on register values automatically, when they are read out of the heap on the target machine.

8.2.3 Uses for process migration

Process migration may be applied to various higher-level concepts to simplify application development. Its primary benefit is in fault-tolerant computing, where each part of a distributed computation can take periodic checkpoints which are used to recover the computation when a machine fails. It can be used in many other applications, a few of which are described below.

Load-balancing is an important issue in a distributed system, especially if the nodes or computation processes are heterogeneous. A system monitor can detect nodes in the system which are overloaded relative to other nodes, and migrate some of the processes off the overloaded node to other nodes carrying a lighter load. Since processes do not observe the result of a migration, this can be done without any special support by the processes.

---

\(^9\) As a future optimization, we may include support for migrating executable code when the architecture is known. Such an optimization would compromise program safety however, since it is not trivial to verify the correctness of assembly code.
Migration can be used to pass “active” messages, or messages which include executable code and data. To pass a message from one process to another, the process may start a new thread, and migrate the thread (including all necessary state) to the machine hosting the other process.

This can be particularly useful for database queries. When the database is hosted on a remote machine, it is highly inefficient to run a query on the local machine that transfers effectively the entire database over the network to extract the few records which the query is interested in. Mainstream database systems work around this by supporting a query language used to describe the query. The local machine sends the text of the query, which is then run on the server. However, these languages are highly specialised and may not be sufficiently expressive for certain inquiries. With process migration, we can migrate an arbitrary program to the remote machine, allowing far greater expressiveness using a more general mechanism.

Similar to active messages, migration can be used to create active files, or files which contain data but also code to manipulate the data, similar to existing file formats which include a macro or scripting language. A simple example would be a multimedia file which includes the program required to play the media.

8.3 Atomic operations and transactions

In databases, transactions play a key role in ensuring that sequences of operations that are run simultaneously do not interfere. Semantically, transactional execution appears atomic; that is, either all the operations in a transaction must succeed, or none of them will succeed. The FIR provides a generalization of transactions for expressing rollback of a distributed computation which is more efficient than using process migration alone in the event of a machine failure.

The primary obstacle in implementing atomic transactions is restoration of the program state. When a transaction is aborted, the entire process state, including all variable and heap values, must be restored to the state it had on entry into the transaction.

Rollback can be expressed with process migration by having a process write a new checkpoint file each time it enters a new atomic section. If the transaction is aborted, the previous state can be restored by restoring the process from a checkpoint. However, since the migration mechanism recompiles the program, and the entire process state must be reconstructed, this operation can be very expensive. Even taking the checkpoint is expensive, since the entire state must be written to a file, even parts of the state that have not changed since a prior checkpoint. By contrast, atomic transactions use a copy-on-write mechanism to keep track of modified state that must be restored if the transaction is rolled back.

The FIR provides three primitives for managing atomic transactions: entry, which enters a new atomic level; commit, which marks an atomic level as completed; and rollback, which aborts all changes made by a particular level and resumes execution at the point where the level was previously entered.

8.3.1 Using atomic transactions in the FIR

Like process migration, atomic transactions use the special-call mechanism in the FIR. Atomic transactions may be nested; each entry operation enters a new atomic level nested within the previous level. Atomic levels are numbered from 1 to N, where 1 is the oldest atomic level entered and N is the most recent. A process that has not entered any atomic transactions is at level 0. A level l keeps track of all changes made

\[10\text{In this paper, we do not consider rollback of I/O operations. However, the concepts discussed here can be extended to include I/O operations with some assistance from the operating system. These extensions will be discussed in a future paper.}\]
to the state that have occurred since \( l \) was entered. Atomic levels use copy-on-write semantics; when a block in the heap is modified, the block is cloned and the pointer table updated to point to the new copy of the block, preserving the data in the original block. On a commit or rollback operation of \( l \), exactly one of these blocks will be discarded.

The primitive for entry is the atomic \( f(c,a_1,\ldots,a_n) \) special-call. \( c \) is an integer that is passed as the first argument to \( f \); on a rollback, the value of \( c \) passed to \( f \) may be changed to indicate that the rollback occurred. This is the only way to carry state information across a rollback\(^{11}\).

The primitive for commit is \( \text{commit}[\text{level}] f(a_1,\ldots,a_n) \). This will commit data for \( \text{level} \) by folding all changes from that level into its previous level. \( \text{level} \) must be in the interval \( \{0 \ldots N\} \), otherwise a runtime exception will occur. If \( \text{level} = 0 \), then the most recent level \( N \) is committed.

The primitive for rollback is \( \text{rollback}[\text{level},c] \). This will revert all changes made by in level \( \text{level} \) and all later levels. \( \text{level} \) must be in the interval \( \{0 \ldots N\} \), otherwise a runtime exception will occur. If \( \text{level} = 0 \), then the most recent level \( N \) is reverted.

Rollback resumes execution at the point where \( \text{level} \) was entered. No function or argument list is specified; the function that was associated with level \( \text{level} \) is saved as part of the checkpoint, to be called with the original atom arguments but with the new value for \( c \). This version of the primitive is a retry primitive; atomic level \( \text{level} \) is automatically re-entered after it (and all later levels) have been rolled back\(^{12}\).

### 8.3.2 Implementation of atomic transactions

Transactions are implemented in close cooperation with the garbage collector. The heap layout is shown in Figure 8.2, which is drawn with the base of the heap at the top of the figure, and the limit of the heap at the bottom.

The heap has the following properties.

- Each heap generation \( i \) is delimited by a base pointer \( \text{base}[i] \), and a limit pointer \( \text{base}[i+1] \), or, in the case of the youngest generation, \( \text{limit} \).
- The upper bound of atomic level \( i \) is delimited by the \( \text{level}[i] \) pointer.
  - \textbf{Invariant}: (Atomic Invariant) all heap data for atomic level \( i \) is between \( \text{base}[i] \) and \( \text{level}[i] \), and it is immutable.

The generational bounds and the atomic level bounds are independent. An atomic level may cross a generational boundary, and often does. The two are related however: since the data in an atomic level is immutable, garbage collection on an atomic level is idempotent.

### Garbage collection

The garbage collector uses generational, mark-sweep, compacting, collection. During the mark phase for generation \( i \), the entire live set for generation \( i \) and all younger generations is traversed, and the live blocks

\(^{11}\)For technical reasons, \( c \) must be an integer in the current implementation. Future implementations will allow \( c \) to be a pointer to a block in the heap which is preserved across a rollback operation.

\(^{12}\)In effect, the state that is captured and restored is the state immediately after level \( \text{level} \) was entered.
are marked. The marking algorithm uses a pointer-reversal scheme to eliminate the need for additional storage during traversal.

During the sweep phase for generation $i$, the heap from $base[i]$ to $limit$ is scanned. If a dead block is encountered, its entry in the pointer table is deleted. Otherwise, if the block is live, it is copied left to the lowest unallocated location in the heap, and the pointer table is updated with the new location. Note that the heap data itself is not modified during collection.

Block ordering is preserved by the collection, and the heap pointers $level[i]$, $base[i]$, $current$, and $limit$ are updated after an allocation to point to their new locations.

**Transaction support**

On atomic entry, a new generation is set up in the heap by creating a new $level[i]$ pointing at $current$, the end of the heap. All data before $current$ becomes immutable. In addition to the $level[i]$ bounds, each level has its own pointer table $pdiff[i]$, that can be used to restore the pointer table if the transaction is aborted. To minimize storage requirements, the $pdiff[i]$ table is stored as a set of differences with the current pointer table $ptable$.

Within a transaction, the only operations requiring special support are the assignment operations. If a block is to be mutated, and the block belongs to a previous generation (its address is below the current $level[i]$), the block is copied into the minor heap, the current pointer table $ptable$ is updated with the new location, and the previous pointer table $pdiff[i-1]$ is updated with the block’s original location. The original data remains unmodified. The garbage collector includes the $pdiff$ tables as “root” pointers; the original block remains live.
When an atomic level \( i \) is committed, the pointer \( \text{level}[i] \) and the difference table \( \text{pdiff}[i] \) are deleted. In general, this will release storage that was needed in case of rollback, and the space is automatically reclaimed during the next garbage collection. On a rollback to atomic level \( i \), the pointer table is restored from the current pointer table and the \( \text{pdiff}[i] \) table, which is deleted along with the \( \text{level}[i] \) delimiter.

### 8.3.3 Using atomic transactions for traditional transactions

When using traditional transactions, a transaction cannot be committed until all nested transactions have been committed. In this case, the commit operation will always be called with \( \text{level} = 0 \). This behaviour is consistent with traditional transactions, where a nested transaction must always be committed before an older transaction can be committed.

### 8.3.4 Using atomic transactions for fault-tolerant computing

In a distributed computation, the processes involved must synchronize periodically. Ideally, the processes would speculatively continue computing across a synchronization point, even if they have not received confirmation that all other processes have reached the synchronization point. In this case, a process can enter a new atomic transaction as it passes the synchronization point and continue computing. If a process in the system fails, the remaining processes can agree on the last known-good synchronization point using a consensus algorithm, and then roll back to that synchronization point to continue the computation.

In this case, the process will monitor where every other process in the system is, and commit a transaction for a synchronization point once it knows all other processes in the system have reached that point. In this scenario, we will always be committing the oldest transaction in the system, which is opposite the behaviour we would expect in traditional distributed transactions. Because of this case, it is useful to generalize the transactions so a process can commit any transaction. When a level is committed, it is simply folded into the previous level so that a later rollback on the previous level will revert changes associated with the committed level as well.

### 8.4 An implementation of fault-tolerant distributed computing

Migration and atomic transactions can be used in conjunction to implement an efficient fault-tolerant distributed computation. In a distributed computation, it is often necessary that the processes synchronize periodically and come to consensus on the current state of the computation. Using traditional programming techniques, the computation on each node must be halted until all processes reach the synchronization point and agree on the current state of the global computation. This imposes serialization in the system and can impact performance if the nodes are heterogeneous with respect to computing speed.

Migration and atomic transactions can be used to support speculative computing each time a synchronization point is reached. When a process reaches a synchronization point, it sends a control message to other processes indicating it has reached the synchronization point. Then the process establishes a new atomic level and continues computing, assuming that other processes will eventually reach the same synchronization point. Periodically, the process uses the migration system to write a full checkpoint of the process to a shared fault-tolerant storage system. Once the process receives confirmation from all other processes that the synchronization point has been reached, it can commit the associated atomic level. It can also delete a
checkpoint file once all other processes have reached the synchronization point and a more recent checkpoint is written for the process.

In the event of a failure in the system (which will likely terminate at least one process abnormally), the system can recover to the last known-valid state by rolling back each surviving process to the appropriate atomic level. The last checkpoint of a terminated process can be resurrected on any surviving machine in the system, using the migration system’s resurrect program. Atomic transactions provide an efficient method to rollback the state of surviving processes, and migration checkpoints provide a method to revive any process that is abnormally terminated during a failure.

Work is in progress to implement this system using the existing primitives.
Bibliography


