Process Migration and Transactions Using a Novel Intermediate Language[1]

Caltech Technical Report caltechCSTR:2002.007

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

November 16, 2002

[1]This technical report is based on version 1.76 of the FIR.

Contents

List of Figures

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.

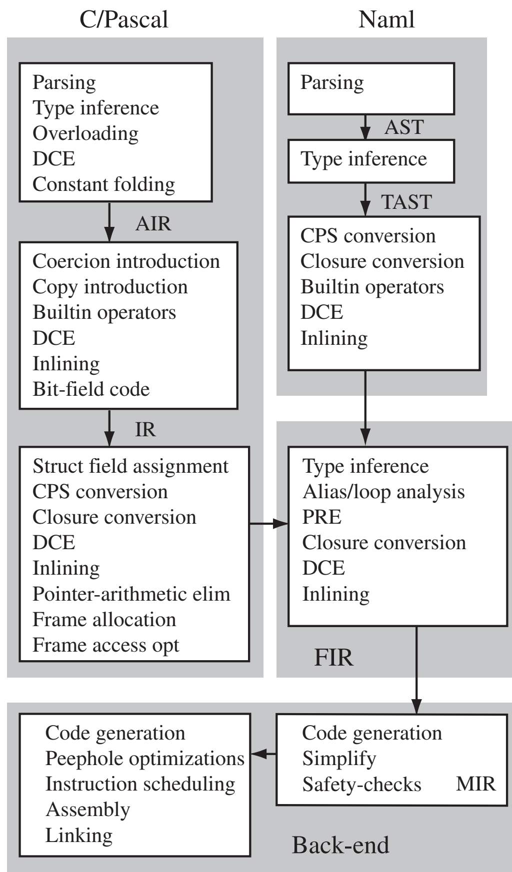

Figure 1.1: Architecture of the Mojave Compiler

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 α, β, γ and the meta-variable tv refer to type variables.

We use the notation $\lceil v_i \rceil_1^n$ to denote the vector $v_1, \ldots, v_n$. The index variable is implicitly i. An alternate notation $\lceil v_j \rceil_{j=1}^n$ 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.

Entity			Description	FIR Entity
		$v_1, v_2, \ldots$	Variable names	Fir.var
		$tv_1, tv_2, \ldots$		
		$\alpha, \beta, \ldots$	Type variables	Fir.ty_var
		$l_1, l_2, \ldots$	Labels	Fir.label
i	::=	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	Integer constants	int
pre	::=	$8 \mid 16 \mid 32 \mid 64$	Integer precisions	Int8 | Int16 | Int32 | Int64
$sign$	::=	**signed** | **unsigned**	Integer signedness	true | false
r	::=	$(i, pre, sign)$	Raw integer constants	Rawint.rawint
$fpre$	::=	$32 \mid 64 \mid 80$	Floating-point precisions	Single | Double | LongDouble
x	::=	$float$	Floating-point constants	Rawfloat.rawfloat
set_I	::=	$[i_1^1, i_2^1], \ldots, [i_1^n, i_2^n]$	Integer interval set	IntSet.t
set_R	::=	$[r_1^1, r_2^1], \ldots, [r_1^n, r_2^n]$	Raw integer interval set	RawIntSet.t
set	::=	$set_I \mid set_R$	Arbitrary sets	IntSet set_I | RawIntSet set_R
$tuple\_class$	::=	**normal** | **box**	Tuple classes	NormalTuple | BoxTuple

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 $\mathbf{enum}_i$ includes the integers $\{0, \ldots, i-1\}$. The maximum enumeration value is bounded by the value max_enum[3]. The type $\mathbf{enum}_0$ is valid and represents the void type. The native integer type $\mathbb{Z}_{pre}^{sign}$ includes values of a specified bit precision and signedness interpretation. The type $\mathbb{R}_{fpre}$ includes floating-point values of a specified bit precision.

Tuples The tuple type $\langle t_1, \ldots, t_n \rangle_{tuple\_class}$ 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 $\textbf{union}(tv[u_1, \ldots, u_m], set_I)$ represents values in a polymorphic union type, which is defined as $tv = \Lambda\alpha_1, \ldots, \alpha_m.\vec{t_1} + \cdots + \vec{t_n}$ (type definitions are discussed below). A value of type $\textbf{union}(tv[u_1, \ldots, u_m], \{j\})$ has type $\vec{t_j}[u_1/\alpha_1, \ldots, u_m/\alpha_m]$[4].

Unsafe data areas The unsafe types **data** and $\textbf{frame}(tv[t_1, \ldots, t_m])$ represent arbitrary data. Values of type **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 **data** type have no explicit substructure, while values with the $\textbf{frame}(tv[t_1, \ldots, t_m])$ type are defined as unsafe records, where the fields in the record are not strongly typed.

Functions The function type $(t_1, \ldots, t_n) \to t$ 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 $tv[t_1, \ldots, t_m]$ applies arguments to a type function, defined as part of a program context (see Chapter 3). If the definition is a parameterized type $\Lambda\alpha_1, \ldots, \alpha_m.t$, the type $tv[t_1, \ldots, t_m]$ is defined as the type $t[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. For example, in a context containing the definition $\gamma = \Lambda\alpha, \beta.\langle\alpha, \beta\rangle_{\textbf{normal}}$, the type $\gamma[\mathbb{Z}_{31}, \textbf{enum}_2]$ is the same as the type $\langle\mathbb{Z}_{31}, \textbf{enum}_2\rangle_{\textbf{normal}}$.

Universal types The universal type $\forall\alpha_1, \ldots, \alpha_m.t$ defines a polymorphic type, where t must be a function type (see Section 4.6).

Existential types The existential type $\exists\alpha_1, \ldots, \alpha_m.t$ defines a type abstraction. The values in an existential type have the form $\textbf{ty\_pack}[\exists\alpha_1, \ldots, \alpha_m.t](v, t_1, \ldots, t_m)$, where v has type $t[t_1/\alpha_1, \ldots, t_m/\alpha_m]$.

Projections The type projection $v.i$ is used for values having existential type $\exists\alpha_0, \ldots, \alpha_{m-1}.t$. If a value $v = \textbf{ty\_pack}[\exists\alpha_0, \ldots, \alpha_{m-1}.t](v', t_0, \ldots, t_{m-1})$ has type $\exists\alpha_0, \ldots, \alpha_{m-1}.t$, 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 *tydef* has the form $\Lambda\alpha_1, \ldots, \alpha_m.tydef_s$, where $tydef_s$ is a simple type definition abstracted over type parameters $\alpha_1, \ldots, \alpha_m$.

A simple type definition can be a type, a disjoint union, or a frame record type. Disjoint unions are defined on type vectors $\vec{t_1} + \cdots + \vec{t_m}$. Each type vector $\vec{t_i}$ is effectively a tuple space $\langle t_1, \ldots, t_n\rangle$. The translation of ML datatypes is straightforward. For example, an ML datatype for labeled binary trees might be declared as follows.

```
type 'a tree =
   Leaf
 | Node of 'a tree * 'a * 'a tree
```

[4] By $\tau[t/\alpha]$ we mean non-capturing substitution of t for α in a term τ. See Section 3.5.3.

		Type	Description	FIR Type
t	$::=$	$\mathbb{Z}_{31}$	Boxed integers	TyInt
	$\mid$	$\mathbf{enum}_i$	Integer enumerations	$\mathsf{TyEnum}\ i$
	$\mid$	$\mathbb{Z}_{pre}^{sign}$	Native integers	$\mathsf{TyRawInt}\ (pre, sign)$
	$\mid$	$\mathbb{R}_{fpre}$	Floating-point values	$\mathsf{TyFloat}\ fpre$
	$\mid$	$\langle t_1, \ldots, t_m \rangle_{tuple\_class}$	Tuple type	$\mathsf{TyTuple}\ (tuple\_class, [t_1; \cdots; t_m])$
	$\mid$	t **array**	Array type	$\mathsf{TyArray}\ t$
	$\mid$	$\mathbf{union}(tv[t_1, \ldots, t_m], set_I)$	Constructor type	$\mathsf{TyUnion}\ (tv, [t_1; \cdots; t_m], set_I)$
	$\mid$	**data**	Unsafe data	$\mathsf{TyRawData}$
	$\mid$	$\mathbf{frame}(tv[t_1, \ldots, t_m])$	Frame record	$\mathsf{TyFrame}\ (tv, [t_1; \cdots; t_m])$
	$\mid$	$(t_1, \ldots, t_m) \to t$	Function type	$\mathsf{TyFun}\ ([t_1; \cdots; t_m], t)$
	$\mid$	$\alpha, \beta, \ldots$	Polymorphic type vars	$\mathsf{TyVar}\ \alpha, \mathsf{TyVar}\ \beta, \ldots$
	$\mid$	$tv[t_1, \ldots, t_m]$	Type application	$\mathsf{TyApply}\ (tv, [t_1; \cdots; t_m])$
	$\mid$	$\forall \alpha_1, \ldots, \alpha_m.t$	Universal types	$\mathsf{TyAll}\ ([\alpha_1; \cdots; \alpha_m], t)$
	$\mid$	$\exists \alpha_1, \ldots, \alpha_m.t$	Existential types	$\mathsf{TyExists}\ ([\alpha_1; \cdots; \alpha_m], t)$
	$\mid$	$v.i$	Abstract type	$\mathsf{TyProject}\ (v, i)$

Figure 2.2: The FIR type system

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

$$tree = \Lambda\alpha.\langle\rangle + \langle tree[\alpha], \alpha, tree[\alpha]\rangle$$

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).

		Type	Description
$\vec{t}$	$::=$	$\langle t_1, \ldots, t_n \rangle$	List of types
R	$::=$	$\left\{ l_1^{sf} : t_1, \ldots, l_m^{sf} : t_m \right\}$	Record type
$tydef_s$	$::=$	t	Type abstraction
	$\mid$	$\vec{t}_1 + \cdots + \vec{t}_m$	Union type
	$\mid$	$\left\{ l_1^{f} : R_1, \ldots, l_n^{f} : R_n \right\}$	Frame type
$tydef$	$::=$	$\Lambda\alpha_1, \ldots, \alpha_m.tydef_s$	Polymorphic type

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

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

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

Similarly, the atom $\mathbf{sizeof}(tv_1, \ldots, tv_n, r)$ 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 $\mathbf{ty\_apply}[t](v, t_1, \ldots, t_m)$ 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 $\mathbf{ty\_pack}[t](v, t_1, \ldots, t_m)$ 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 $\mathbf{ty\_unpack}(v)$ 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 $unop\, a$, and binary operations a_1 $binop$ a_2. The operator tables are listed in Figures 2.5–2.7.

[5] A notable arithmetic exception is division by zero.

		Definition	Description	FIR Expression
a	$::=$	$\mathbf{enum}_{i_{bound}}(i_{value})$	Enumerated value	AtomEnum (i_{bound}, i_{value})
	$\mid$	$\mathbf{int}(i)$	Boxed integer constants	AtomInt i
	$\mid$	$\mathbf{rawint}^{sign}_{pre}(r)$	Raw integer constants	AtomRawInt r
	$\mid$	$\mathbf{float}_{fpre}(x)$	Floating-point constants	AtomFloat x
	$\mid$	$\mathbf{label}(tv, l^f, l^{sf}, r)$	Offset into a frame record	AtomLabel (tv, l^f, l^{sf}, r)
	$\mid$	$\mathbf{sizeof}(tv_1, \ldots, tv_n, r)$	Size of frames, plus r	AtomSizeof $([tv_1; \cdots; tv_n], r)$
	$\mid$	$\mathbf{const}[t](tv, i)$	Constant union constructor	AtomConst (t, tv, i)
	$\mid$	v	Variables	AtomVar v
	$\mid$	$\mathbf{ty\_apply}[t](a, t_1, \ldots, t_m)$	Type application	AtomTyApply $(a, t, [t_1; \cdots; t_m])$
	$\mid$	$\mathbf{ty\_pack}[t](v, t_1, \ldots, t_m)$	Existential packing	AtomTyPack $(v, t, [t_1; \cdots; t_m])$
	$\mid$	$\mathbf{ty\_unpack}(v)$	Existential unpacking	AtomTyUnpack v
	$\mid$	$\mathit{unop}\, a$	Unary operation	AtomUnop (unop, a)
	$\mid$	$a_1\ \mathit{binop}\ a_2$	Binary operation	AtomBinop $(\mathit{binop}, a_1, a_2)$

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 *unop*, the term $\mathbf{arg}(\mathit{unop})$ specifies the type of the argument of the operator (in the *Arg* column), and the term $\mathbf{res}(\mathit{unop})$ specifies the type of the result (in the *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.

Operator	Result	Arg	Description	FIR Entity
$\sim\!-_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Unary negation	UMinusIntOp
$\sim\!-_{\mathbb{Z}^{sign}_{pre}}$	$\mathbb{Z}^{sign}_{pre}$	$\mathbb{Z}^{sign}_{pre}$		UMinusRawIntOp $(\mathit{pre}, \mathit{sign})$
$\sim\!-_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		UMinusFloatOp fpre
$\mathbf{lnot}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Bitwise negation	NotIntOp
$\mathbf{lnot}_{\mathbb{Z}^{sign}_{pre}}$	$\mathbb{Z}^{sign}_{pre}$	$\mathbb{Z}^{sign}_{pre}$		NotRawIntOp $(\mathit{pre}, \mathit{sign})$
$\mathbf{bitfield}^{i,j}_{pre,sign}$	$\mathbb{Z}^{sign}_{pre}$	$\mathbb{Z}^{sign}_{32}$	Extract bitfield at offset i (size j)	RawBitFieldOp $(\mathit{pre}, \mathit{sign}, i, j)$
$\mathbf{abs}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	Absolute value	AbsFloatOp fpre
$\mathbf{sin}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	Sine	SinOp fpre
$\mathbf{cos}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	Cosine	CosOp fpre
$\mathbf{sqrt}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	Square root	SqrtOp fpre
$(t_d \leftarrow t_s)$	t_d	t_s	Coerce between types (safely)	...

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 *binop*, the term $\mathbf{arg}_i(binop)$ specifies the type of argument i of the operator. The index i is either 1 or 2, and the type is listed in the *Arg i* column. The term $\mathbf{res}(binop)$ specifies the type of the result, listed in the *Result* column.

The comparison operators are shown in Figure 2.6. In most cases, the result of a comparison has the type $\mathbf{enum}_2$, which is used to represent Boolean values. The $\mathbf{cmp}_p$ operators are an exception. The result of a comparison $i\ \mathbf{cmp}_p\ j$ 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 $=_t$ compares two values of type t. The type t is arbitrary, and we do not specify the semantics precisely. For any comparison $a_1 =_t a_2$, the result is not true if $a_1 \neq a_2$. The comparison $a_1 \neq_t a_2$ is the same as $\neg(a_1 =_t a_2)$.

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

Operator	Result	Arg 1	Arg 2	Description	FIR Entity
$=_{\mathbb{Z}_{31}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Test for equality	EqIntOp
$=_{\mathbb{Z}_{pre}^{sign}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		EqRawIntOp $(pre, sign)$
$=_{\mathbb{R}_{fpre}}$	$\mathbf{enum}_2$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		EqFloatOp *fpre*
$\neq_{\mathbb{Z}_{31}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Test for inequality	NeqIntOp
$\neq_{\mathbb{Z}_{pre}^{sign}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		NeqRawIntOp $(pre, sign)$
$\neq_{\mathbb{R}_{fpre}}$	$\mathbf{enum}_2$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		NeqFloatOp *fpre*
$<_{\mathbb{Z}_{31}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Test for less than	LtIntOp
$<_{\mathbb{Z}_{pre}^{sign}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		LtRawIntOp $(pre, sign)$
$<_{\mathbb{R}_{fpre}}$	$\mathbf{enum}_2$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		LtFloatOp *fpre*
$\leq_{\mathbb{Z}_{31}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Test for less than or equal	LeIntOp
$\leq_{\mathbb{Z}_{pre}^{sign}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		LeRawIntOp $(pre, sign)$
$\leq_{\mathbb{R}_{fpre}}$	$\mathbf{enum}_2$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		LeFloatOp *fpre*
$>_{\mathbb{Z}_{31}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Test for greater than	GtIntOp
$>_{\mathbb{Z}_{pre}^{sign}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		GtRawIntOp $(pre, sign)$
$>_{\mathbb{R}_{fpre}}$	$\mathbf{enum}_2$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		GtFloatOp *fpre*
$\geq_{\mathbb{Z}_{31}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Test for greater than or equal	GeIntOp
$\geq_{\mathbb{Z}_{pre}^{sign}}$	$\mathbf{enum}_2$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		GeRawIntOp $(pre, sign)$
$\geq_{\mathbb{R}_{fpre}}$	$\mathbf{enum}_2$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		GeFloatOp *fpre*
$\mathbf{cmp}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	ML-style comparison	CmpIntOp
$\mathbf{cmp}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		CmpRawIntOp $(pre, sign)$
$\mathbf{cmp}_{\mathbb{R}_{fpre}}$	$\mathbb{Z}_{31}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		CmpFloatOp *fpre*
$=_t$	$\mathbf{enum}_2$	t	t	Test for pointer equality	EqEqOp t
$\neq_t$	$\mathbf{enum}_2$	t	t	Test for pointer inequality	NeqEqOp t

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.

Operator	Result	Arg 1	Arg 2	Description	FIR Entity
$+_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Addition	PlusIntOp
$+_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		PlusRawIntOp $(pre, sign)$
$+_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		PlusFloatOp $fpre$
$-_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Subtraction	MinusIntOp
$-_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		MinusRawIntOp $(pre, sign)$
$-_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		MinusFloatOp $fpre$
$*_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Multiplication	MulIntOp
$*_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		MulRawIntOp $(pre, sign)$
$*_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		MulFloatOp $fpre$
$/_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Division	DivIntOp
$/_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		DivRawIntOp $(pre, sign)$
$/_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		DivFloatOp $fpre$
$\mathbf{rem}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Remainder	RemIntOp
$\mathbf{rem}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		RemRawIntOp $(pre, sign)$
$\mathbf{rem}_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		RemFloatOp $fpre$
lsl	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Logical shift left	LslIntOp
lsr	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Logical shift right	LsrIntOp
asr	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Arithmetic shift right	AsrIntOp
$\mathbf{sl}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	Shift left	SlRawIntOp $(pre, sign)$
$\mathbf{sr}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	Shift right	SrRawIntOp $(pre, sign)$
$\mathbf{and}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Logical and	AndIntOp
$\mathbf{and}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		AndRawIntOp $(pre, sign)$
$\mathbf{and}_{\mathbf{enum}_i}$	$\mathbf{enum}_i$	$\mathbf{enum}_i$	$\mathbf{enum}_i$		AndEnumOp i
$\mathbf{or}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Logical or	OrIntOp
$\mathbf{or}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		OrRawIntOp $(pre, sign)$
$\mathbf{or}_{\mathbf{enum}_i}$	$\mathbf{enum}_i$	$\mathbf{enum}_i$	$\mathbf{enum}_i$		OrEnumOp i
$\mathbf{xor}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Logical xor	XorIntOp
$\mathbf{xor}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		XorRawIntOp $(pre, sign)$
$\mathbf{xor}_{\mathbf{enum}_i}$	$\mathbf{enum}_i$	$\mathbf{enum}_i$	$\mathbf{enum}_i$		XorEnumOp i
$\mathbf{max}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Maximum of two values	MaxIntOp
$\mathbf{max}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		MaxRawIntOp $(pre, sign)$
$\mathbf{max}_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		MaxFloatOp $fpre$
$\mathbf{min}_{\mathbb{Z}_{31}}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$	Minimum of two values	MinIntOp
$\mathbf{min}_{\mathbb{Z}_{pre}^{sign}}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$		MinRawIntOp $(pre, sign)$
$\mathbf{min}_{\mathbb{R}_{fpre}}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$		MinFloatOp $fpre$
$\mathbf{atan2}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$	Arc tangent	Atan2Op $fpre$
$\mathbf{set\_bitfield}_{pre,sign}^{i,j}$	$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{32}^{sign}$	$\mathbb{Z}_{pre}^{sign}$	Set bit field at offset i	RawSetBitFieldOp $(\ldots)$

Figure 2.7: FIR binary operators

LetAtom The **let** $v: t = a$ **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** $v: t_v = (\text{``}s\text{''} : t_s)(a_1, \ldots, a_n)$ **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 $t_s = (u_1, \ldots, u_n) \rightarrow t_v$, 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 $(u_1, \ldots, u_n) \rightarrow \mathbf{enum}_0$, and each argument a_i must have type u_i.

The return type of the function is the empty type $\mathbf{enum}_0$. 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 **migrate** $[i, a_p, a_o]$ $a_{fun}(a_1, \ldots, a_n)$ defines a process migration. The argument (a_p, a_o) specifies the destination (for example, the name of a migration server), and the argument $a_{fun}(a_1, \ldots, a_n)$ specifies a function to be called once the process migrates, along with its arguments.
- The operator **atomic** $a_{fun}(a_{const}, a_1, \ldots, a_n)$ 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_{fun} is called with arguments $a_{const}, a_1, \ldots, a_n$.
- The operator **rollback** $[a_{level}, a_{const}]$ is used to abort a transaction. The atom a_{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_{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 **atomic** $a_{fun}(a_{const}, a_1, \ldots, a_n)$, and the transaction is aborted with the **rollback** $[a'_{level}, a'_{const}]$, the transaction is *resumed* with the tail-call $a_{fun}(a'_{const}, a_1, \ldots, a_n)$.
- The operator **commit** $[a_{level}]$ $a_{fun}(a_1, \ldots, a_n)$ commits the transaction identified by a_{level}. The transaction checkpoint identified by a_{level} is discarded, and the function a_{fun} is called with arguments $a_1, \ldots, a_n$.

Match The match statement **match** a **with** $\lceil s_i \mapsto e_i \rceil_1^n$ 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 **let** $v =$ *alloc* **in** e expression allocates a data aggregate, using one of the *alloc* forms shown in Figure 2.9.

Values are projected from an aggregate data area using the **let** $v\colon t = a_1[a_2]$ **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 $a_1[a_2]\colon t \leftarrow a_3;\ e$ 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 $\mathbf{let}\, v\colon t = \mathbf{global}\, l\ \mathbf{in}\ e$, and assigned with the expression $\mathbf{global}\, l\colon t \leftarrow a;\ e$.

		Definition	Description	FIR Expression
e	$::=$	$\mathbf{let}\, v\colon t = a\ \mathbf{in}\ e$	Primitive operations	LetAtom (v, t, a, e)
	$\vert$	$\mathbf{let}\, v\colon t_v = (``s" : t_s)(a_1, \ldots, a_n)\ \mathbf{in}\ e$	Calls to the runtime	LetExt $(v, t_v, s, t_s, [a_1; \cdots; a_n], e))$
	$\vert$	$a(a_1, \ldots, a_n)$	Tail-call	TailCall $(*, a, [a_1; \cdots; a_n]))$
	$\vert$	$\mathbf{special}\ \mathit{tailop}$	Special tail-call	SpecialCall $(*, \mathit{tailop}))$
	$\vert$	$\mathbf{match}\, a\ \mathbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n$	Case analysis	Match $(a, [\ \lceil (*, s_i, e_i) \rceil_1^n\ ])$
	$\vert$	$\mathbf{let}\, v = \mathit{alloc}\ \mathbf{in}\ e$	Allocation	LetAlloc $(v, \mathit{alloc}, e))$
	$\vert$	$\mathbf{let}\, v\colon t = a_1[a_2]\ \mathbf{in}\ e$	Load from heap	LetSubscript $(*, v, t, a_1, a_2, e))$
	$\vert$	$a_1[a_2]\colon t \leftarrow a_3;\ e$	Store into heap	SetSubscript $(*, *, a_1, a_2, t, a_3, e))$
	$\vert$	$\mathbf{let}\, v\colon t = \mathbf{global}\, l\ \mathbf{in}\ e$	Load from global	LetGlobal $(*, v, t, l, e))$
	$\vert$	$\mathbf{global}\, l\colon t \leftarrow a;\ e$	Store into global	SetGlobal $(*, *, l, t, a, e))$

Figure 2.8: FIR expressions

		Definition	Description	FIR Expression
unop	$::=$	$- \mid\ !\ \mid \cdots$	Unary operations	Fir.unop
binop	$::=$	$+ \mid - \mid * \mid / \mid \cdots$	Binary operations	Fir.binop
alloc	$::=$	$\langle a_1, \ldots, a_n \rangle_{tuple\_class} : t$	Tuple allocation	AllocTuple $(\ldots)$
	$\vert$	$\mathbf{union}(tv[a_1, \ldots, a_n], i) : t$	Union allocation	AllocUnion $(\ldots)$
	$\vert$	$\mathbf{array}(a_{size}, a_{init}) : t$	Array allocation	AllocVArray $(\ldots)$
	$\vert$	$\mathbf{malloc}(a) : t$	Rawdata allocation	AllocMalloc $(\ldots)$
	$\vert$	$\mathbf{frame}(tv[t_1, \ldots, t_m])$	Allocate a frame	AllocFrame $(\ldots)$
tailop	$::=$	$\mathbf{migrate}\ [i, a_p, a_o]\ a_{fun}(a_1, \ldots, a_n)$	System migration	TailSysMigrate $(\ldots)$
	$\vert$	$\mathbf{atomic}\ a_{fun}(a_{const}, a_1, \ldots, a_n)$	Transaction entry	TailAtomic $(\ldots)$
	$\vert$	$\mathbf{rollback}\ [a_{level}, a_{const}]$	Transaction rollback	TailAtomicRollback $(\ldots)$
	$\vert$	$\mathbf{commit}\ [a_{level}]\ a_{fun}(a_1, \ldots, a_n)$	Transaction commit	TailAtomicCommit $(\ldots)$

Figure 2.9: FIR operators

Chapter 3

Judgments

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

		Definition	Description
h	::=	$\mathbf{enum}_{i_{bound}}(i_{value})$	Enumerated values
	|	$\mathbf{int}(i)$	Boxed integer constants
	|	$\mathbf{rawint}^{sign}_{pre}(r)$	Raw integer constants
	|	$\mathbf{float}_{fpre}(x)$	Floating-point constants
	|	$\mathbf{label}(tv, l^f, l^r, r)$	Offset into a frame record
	|	v	Variables

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 $\alpha_1, \ldots, \alpha_m$,[1] 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 $\langle \mathrm{C} \rangle$; 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 $\langle \text{C} : \textbf{frame}(tv[t_1, \ldots, t_n])\rangle$, where $\textbf{frame}(tv[t_1, \ldots, t_n])$ is the type of the frame.

		Definition	Description
b	$::=$	h	Heap values
	$\vert$	$\Lambda\alpha_1, \ldots, \alpha_m.\lambda v_1, \ldots, v_n.e$	Functions
	$\vert$	$\textbf{ty\_pack}[t](v, t_1, \ldots, t_m)$	Type packing
	$\vert$	$\langle h_1, \ldots, h_n\rangle$	Tuples and arrays
	$\vert$	$tv_i(h_1, \ldots, h_n)$	Unions
	$\vert$	$\langle \text{C}\rangle$	Raw data
	$\vert$	$\langle \text{C} : \textbf{frame}(tv[t_1, \ldots, t_n])\rangle$	Frame data

Figure 3.2: Store values

3.2 Kinds

The program types are also defined/declared as part of the context Γ. 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 $tydef_s$ as "small" types ω, "large" types Ω, disjoint unions $\omega_{union[n]}$, and frames ω_{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 $k = \omega^m \to k_s$ represents a parameterized type definition *tydef*. The number of parameters m may be any nonnegative integer. If $m = 0$, we often omit the type parameters.

$$\begin{array}{rcl} k_s & ::= & \omega \\ & \vert & \Omega \\ & \vert & \omega_{union[n]} \\ & \vert & \omega_{frame} \\ \\ k & ::= & \omega^m \to k_s \end{array}$$

Figure 3.3: Kinds

3.3 Contexts

A program context Γ 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 $tv\colon k = tydef$ defines a type named tv, having kind k, and value *tydef*.
2. The variable definition $v\colon t = b$ defines a variable named v, with type t and store value b.

3. The global definition $l\colon t = h$ 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.

$$
\begin{array}{rcll}
\mathit{def} & ::= & v\colon t & \text{Variable declaration} \\
& | & v\colon t = b & \text{Variable definition} \\
& | & l\colon t & \text{Global declaration} \\
& | & l\colon t = h & \text{Global definition} \\
& | & tv\colon k & \text{Type declaration} \\
& | & tv\colon k = \mathit{tydef} & \text{Type definition} \\
\\
\Gamma & ::= & \epsilon & \text{Empty environment} \\
& | & \Gamma, \mathit{def} & \text{Adding a definition}
\end{array}
$$

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 $\Gamma \vdash \diamond$ specifies that the context Γ 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 $v\colon t$ and definition $v\colon t = b$, the term t must be a well-formed type, and the value b must have type t. Similarly, all type and global definitions in Γ must be well-formed.

The judgment $\Gamma \vdash \mathit{tydef}_1 = \mathit{tydef}_2 : k$ is a type definition equality judgment. When the judgment is true, tydef_1 and tydef_2 have the specified kind, and they are equal. There is no separate membership judgment $\Gamma \vdash \mathit{tydef} : k$. However, we will often use the shorthand form $\Gamma \vdash \mathit{tydef} : k$ to stand for $\Gamma \vdash \mathit{tydef} = \mathit{tydef} : k$.

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 $\Gamma \vdash \mathit{tailop} : \mathbf{special}\, t$ judgment specifies that *tailop* is a special-call operator returning type t. The $\Gamma \vdash \mathit{alloc} : \mathbf{alloc}\, t$ judgment specifies that *alloc* is an allocation operator for a value of type t. The record judgment $\Gamma \vdash R : \mathbf{record}$ 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.

Judgment	Interpretation
$\Gamma \vdash \diamond$	Context Γ is well-formed
$\Gamma \vdash \mathit{tydef}_1 = \mathit{tydef}_2 : k$	tydef_1 and tydef_2 are equal type definitions (or types)
$\Gamma \vdash tv_1 = tv_2 : k$	tv_1 and tv_2 are equal type definitions
$\Gamma \vdash l : t$	Global label l has type t
$\Gamma \vdash a : t$	Atom a has type t
$\Gamma \vdash b : t$	Store value b has type t
$\Gamma \vdash e : t$	Program e has type t
$\Gamma \vdash \mathit{tailop} : \mathbf{special}\, t$	tailop is a special-call of type t
$\Gamma \vdash \mathit{alloc} : \mathbf{alloc}\, t$	alloc is an allocation of a value of type t
$\Gamma \vdash R : \mathbf{record}$	R is a record type

Figure 3.5: Judgments

3.5.1 Domain of context Γ

The *domain* $\mathit{dom}(\Gamma)$ of a context Γ 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.

$$
\begin{array}{rcl}
\mathit{dom}(\epsilon) & = & \{\} \\
\mathit{dom}(\Gamma, v : *) & = & \mathit{dom}(\Gamma) \cup \{v\} \\
\mathit{dom}(\Gamma, v : * = *) & = & \mathit{dom}(\Gamma) \cup \{v\} \\
\mathit{dom}(\Gamma, l : *) & = & \mathit{dom}(\Gamma) \cup \{l\} \\
\mathit{dom}(\Gamma, l : * = *) & = & \mathit{dom}(\Gamma) \cup \{l\} \\
\mathit{dom}(\Gamma, tv : *) & = & \mathit{dom}(\Gamma) \cup \{tv\} \\
\mathit{dom}(\Gamma, tv : * = *) & = & \mathit{dom}(\Gamma) \cup \{tv\}
\end{array}
$$

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.

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

The free variables of binary operators are defined as follows.

$$
FV(\mathit{binop}) ::= \begin{cases} FV(t) & \text{if } \mathit{binop} = \,=_t \text{ or } \mathit{binop} = \,\neq_t \\ \{\} & \text{otherwise} \end{cases}
$$

t	$FV(t)$
$\mathbb{Z}_{31}$	$\{\}$
$\mathbf{enum}_i$	$\{\}$
$\mathbb{Z}_{pre}^{sign}$	$\{\}$
$\mathbb{R}_{fpre}$	$\{\}$
$\langle t_1, \ldots, t_m \rangle_{tuple\_class}$	$\bigcup_{i=1}^{m} FV(t_i)$
t **array**	$FV(t)$
$\mathbf{union}(tv[t_1, \ldots, t_m], set_I)$	$\{tv\} \cup \bigcup_{i=1}^{m} FV(t_i)$
data	$\{\}$
$\mathbf{frame}(tv[t_1, \ldots, t_m])$	$\{tv\} \cup \bigcup_{i=1}^{m} FV(t_i)$
$(t_1, \ldots, t_m) \to t$	$FV(t) \cup \bigcup_{i=1}^{m} FV(t_i)$
α	$\{\alpha\}$
$tv[t_1, \ldots, t_m]$	$\{tv\} \cup \bigcup_{i=1}^{m} FV(t_i)$
$\forall \alpha_1, \ldots, \alpha_m.t$	$FV(t) - \{\alpha_1, \ldots, \alpha_m\}$
$\exists \alpha_1, \ldots, \alpha_m.t$	$FV(t) - \{\alpha_1, \ldots, \alpha_m\}$
$v.i$	$\{v\}$

Figure 3.6: Type free variables

t	$FV(t)$
$\langle t_1, \ldots, t_n \rangle$	$\bigcup_{i=1}^{n} FV(t_i)$
$\vec{t}_1 + \cdots + \vec{t}_n$	$\bigcup_{i=1}^{n} FV(\vec{t})$
$\{l_1^r : t_1, \ldots, l_n^r : t_n\}$	$\bigcup_{i=1}^{n} FV(t_i)$
$\left\{l_1^f : R_1, \ldots, l_n^f : R_n\right\}$	$\bigcup_{i=1}^{n} FV(R_i)$
$\Lambda \alpha_1, \ldots, \alpha_m.tydef_s$	$FV(tydef_s) - \{\alpha_1, \ldots, \alpha_m\}$

Figure 3.7: Type definition free variables

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 Γ are the free variables of all the definitions, excluding the variables being declared.

$$FV(\Gamma) = \left(\bigcup_{d \in \Gamma} FV(d) \right) - dom(\Gamma)$$

a	$FV(a)$
$\mathbf{enum}_{i_{bound}}(i_{value})$	$\{\}$
$\mathbf{int}(i)$	$\{\}$
$\mathbf{rawint}_{pre}^{sign}(r)$	$\{\}$
$\mathbf{float}_{fpre}(x)$	$\{\}$
$\mathbf{label}(tv, l^f, l^r, r)$	$\{tv\}$
$\mathbf{sizeof}(tv_1, \ldots, tv_n, r)$	$\{tv_1, \ldots, tv_n\}$
$\mathbf{const}[t](tv, i)$	$\{tv\} \cup FV(t)$
v	$\{v\}$
$\mathbf{ty\_apply}[t](a, t_1, \ldots, t_m)$	$FV(t) \cup FV(a) \cup \bigcup_{i=1}^m FV(t_i)$
$\mathbf{ty\_pack}[t](v, t_1, \ldots, t_m)$	$FV(t) \cup \{v\} \cup \bigcup_{i=1}^m FV(t_i)$
$\mathbf{ty\_unpack}(v)$	$\{v\}$
$unop\ a$	$FV(a) \cup FV(unop)$
$a_1\ binop\ a_2$	$FV(a_1) \cup FV(a_2) \cup FV(binop)$

Figure 3.8: Atom free variables

e	$FV(e)$
$\mathbf{let}\, v : t = a\ \mathbf{in}\ e$	$FV(t) \cup FV(a) \cup (FV(e) - \{v\})$
$\mathbf{let}\, v : t_v = (\text{“}s\text{”} : t_s)(a_1, \ldots, a_n)\ \mathbf{in}\ e$	$FV(t_v) \cup FV(t_s) \cup (\bigcup_{i=1}^n FV(a_i)) \cup (FV(e) - \{v\})$
$a(a_1, \ldots, a_n)$	$FV(a) \cup \bigcup_{i=1}^n FV(a_i)$
$\mathbf{special}\ tailop$	$FV(tailop)$
$\mathbf{match}\, a\ \mathbf{with}\ s_1 \mapsto e_1, \ldots, s_n \mapsto e_n$	$FV(a) \cup \bigcup_{i=1}^n FV(e_i)$
$\mathbf{let}\, v = alloc\ \mathbf{in}\ e$	$FV(alloc) \cup (FV(e) - \{v\})$
$\mathbf{let}\, v : t = a_1[a_2]\ \mathbf{in}\ e$	$FV(t) \cup FV(a_1) \cup FV(a_2) \cup (FV(e) - \{v\})$
$a_1[a_2] : t \leftarrow a_3;\ e$	$FV(a_1) \cup FV(a_2) \cup FV(t) \cup FV(a_3) \cup FV(e)$
$\mathbf{let}\, v : t = \mathbf{global}\, l\ \mathbf{in}\ e$	$FV(t) \cup \{l\} \cup (FV(e) - \{v\})$
$\mathbf{global}\, l : t \leftarrow a;\ e$	$\{l\} \cup FV(t) \cup FV(a) \cup FV(e)$

Figure 3.9: Expression free variables

$alloc$	$FV(alloc)$
$\langle a_1, \ldots, a_n \rangle_{tuple\_class} : t$	$FV(t) \cup \bigcup_{i=1}^n FV(a_i)$
$\mathbf{union}(tv[a_1, \ldots, a_m], i) : t$	$FV(t) \cup \{tv\} \cup \bigcup_{i=1}^m FV(a_i)$
$\mathbf{array}(a_{size}, a_{init}) : t$	$FV(t) \cup FV(a_{size}) \cup FV(a_{init})$
$\mathbf{malloc}(a) : t$	$FV(t) \cup FV(a)$
$\mathbf{frame}(tv[t_1, \ldots, t_m])$	$\{tv\} \cup \bigcup_{i=1}^m FV(t_i)$

Figure 3.10: Allocation operator free variables

tailop	$FV(tailop)$
migrate $[i, a_p, a_o]\ a_{fun}(a_1, \ldots, a_n)$	$FV(a_p) \cup FV(a_o) \cup FV(a_{fun}) \cup \bigcup_{i=1}^{n} FV(a_i)$
atomic $a_{fun}(a_{const}, a_1, \ldots, a_n)$	$FV(a_{fun}) \cup FV(a_{const}) \cup \bigcup_{i=1}^{n} FV(a_i)$
rollback $[a_{level}, a_{const}]$	$FV(a_{level}) \cup FV(a_{const})$
commit $[a_{level}]\ a_{fun}(a_1, \ldots, a_n)$	$FV(a_{level}) \cup FV(a_{fun}) \cup \bigcup_{i=1}^{n} FV(a_i)$

Figure 3.11: Special-call free variables

b	$FV(b)$
h	$FV(h)$
$\Lambda\alpha_1, \ldots, \alpha_m.\lambda v_1, \ldots, v_n.e$	$FV(e) - \{\alpha_1, \ldots, \alpha_m, v_1, \ldots, v_n\}$
$\mathbf{ty\_pack}[t](v, t_1, \ldots, t_m)$	$FV(t) \cup \{v\} \cup \bigcup_{i=1}^{m} FV(t_i)$
$\langle h_1, \ldots, h_n \rangle$	$\bigcup_{i=1}^{n} FV(h_i)$
$tv_i(h_1, \ldots, h_n)$	$\{tv\} \cup \bigcup_{i=1}^{n} FV(h_i)$
$\langle \mathrm{C} \rangle$	$\{\}$
$\langle \mathrm{C} : \mathbf{frame}(tv[t_1, \ldots, t_m]) \rangle$	$\{tv\} \cup \bigcup_{i=1}^{m} FV(t_i)$

Figure 3.12: Store free variables

d	$FV(d)$
$tv : k$	$\{\}$
$v : t$	$FV(t)$
$l : t$	$FV(t)$
$tv : k = tydef$	$FV(tydef)$
$v : t = b$	$FV(t) \cup FV(b)$
$l : t = h$	$FV(t) \cup FV(h)$

Figure 3.13: Definition free variables

3.5.3 Type substitution

Substitution $tydef[u/\alpha]$ is defined over type definitions, substituting a type u for the type variable α in type definition $tydef$. 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.

t	$t[u/\alpha]$
$\mathbb{Z}_{31}$	$\mathbb{Z}_{31}$
$\mathbf{enum}_i$	$\mathbf{enum}_i$
$\mathbb{Z}_{pre}^{sign}$	$\mathbb{Z}_{pre}^{sign}$
$\mathbb{R}_{fpre}$	$\mathbb{R}_{fpre}$
$\langle t_1, \ldots, t_n \rangle_{tuple\_class}$	$\langle t_1[u/\alpha], \ldots, t_n[u/\alpha] \rangle_{tuple\_class}$
t **array**	$t[u/\alpha]$ **array**
$\mathbf{union}(tv[t_1, \ldots, t_m], set_I)$	$\mathbf{union}(tv[t_1[u/\alpha], \ldots, t_m[u/\alpha]], set_I)$
data	**data**
$\mathbf{frame}(tv[t_1, \ldots, t_m])$	$\mathbf{frame}(tv[t_1[u/\alpha], \ldots, t_m[u/\alpha]])$
$(t_1, \ldots, t_n) \to t$	$(t_1[u/\alpha], \ldots, t_n[u/\alpha]) \to t[u/\alpha]$
β	$\begin{cases} u & \text{if } \beta = \alpha \\ \beta & \text{otherwise} \end{cases}$
$tv[t_1, \ldots, t_m]$	$tv[t_1[u/\alpha], \ldots, t_m[u/\alpha]]$
$\forall \beta_1, \ldots, \beta_m.t$	$\begin{cases} \forall \beta_1, \ldots, \beta_m.t & \text{if } \alpha \in \{\beta_1, \ldots, \beta_m\} \\ \forall \beta_1, \ldots, \beta_m.t[u/\alpha] & \text{otherwise} \end{cases}$
$\exists \beta_1, \ldots, \beta_m.t$	$\begin{cases} \exists \beta_1, \ldots, \beta_m.t & \text{if } \alpha \in \{\beta_1, \ldots, \beta_m\} \\ \exists \beta_1, \ldots, \beta_m.t[u/\alpha] & \text{otherwise} \end{cases}$
$v.i$	$v.i$

Figure 3.14: Type substitution

t	$t[u/\alpha]$
$\langle t_1, \ldots, t_n \rangle$	$\langle t_1[u/\alpha], \ldots, t_n[u/\alpha] \rangle$
$\vec{t_1} + \cdots + \vec{t_n}$	$\vec{t_1}[u/\alpha] + \cdots + \vec{t_n}[u/\alpha]$
$\{l_1^r : t_1, \ldots, l_n^r : t_n\}$	$\{l_1^r : t_1[u/\alpha], \ldots, l_n^r : t_n[u/\alpha]\}$
$\left\{l_1^f : R_1, \ldots, l_n^f : R_n\right\}$	$\left\{l_1^f : R_1[u/\alpha], \ldots, l_n^f : R_n[u/\alpha]\right\}$
$\Lambda \beta_1, \ldots, \beta_m.\mathit{tydef}_s$	$\begin{cases} \Lambda \beta_1, \ldots, \beta_m.\mathit{tydef}_s & \text{if } \alpha \in \{\beta_1, \ldots, \beta_m\} \\ \Lambda \beta_1, \ldots, \beta_m.\mathit{tydef}_s[u/\alpha] & \text{otherwise} \end{cases}$

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 $i \in \mathbb{Z}_{31}$) as premises.

Recall that we use the notation $\lceil v_i \rceil_1^n$ to denote the vector $v_1, \ldots, v_n$. The index variable is implicitly i. An alternate notation $\lceil v_j \rceil_{j=1}^n$ explicitly uses a different index variable j. We also use the notation $\lceil u_i/\alpha_i \rceil_1^n$ to represent a vector of type substitutions.

We use the shorthand $\Gamma \vdash \lceil J_i \rceil_1^n$ 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 $\lceil \Gamma_i \vdash J_i \rceil_1^n$ is similar.

We use the notation (Γ_1, Γ_2) **as** Γ to indicate that the context Γ is the same as (Γ_1, Γ_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.

$$\frac{}{\vdash \diamond} \qquad \text{WF-Empty-Context}$$

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 Γ.

$$\frac{\Gamma \vdash J}{\Gamma, tv : k \vdash J} \qquad \text{Thin-Type}$$

$$\frac{\Gamma \vdash J \quad \Gamma \vdash t : \Omega}{\Gamma, v : t \vdash J} \qquad \text{Thin-Var}$$

$$\frac{\Gamma \vdash J \quad \Gamma \vdash t : \Omega}{\Gamma, l : t \vdash J} \qquad \text{Thin-Global}$$

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.

$$\frac{\Gamma, tv\colon k \vdash \mathit{tydef} : k \quad \Gamma, tv\colon k \vdash J}{\Gamma, tv\colon k = \mathit{tydef} \vdash J} \qquad \text{THIN-TYPE-DEF}$$

$$\frac{\Gamma, v\colon t \vdash b : t \quad \Gamma, v\colon t \vdash J}{\Gamma, v\colon t = b \vdash J} \qquad \text{THIN-VAR-DEF}$$

$$\frac{\Gamma, l\colon t \vdash h : t \quad \Gamma, l\colon t \vdash J}{\Gamma, l\colon t = h \vdash J} \qquad \text{THIN-GLOBAL-DEF}$$

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.

$$\frac{\Gamma, tv\colon k \vdash \diamond}{\Gamma, tv\colon k \vdash tv = tv : k} \qquad \text{TY-REF}$$

The symmetric and transitive forms have the usual definitions.

$$\frac{\Gamma \vdash t_2 = t_1 : k}{\Gamma \vdash t_1 = t_2 : k} \qquad \text{TY-SYM}$$

$$\frac{\Gamma \vdash t_1 = t_2 : k \quad \Gamma \vdash t_2 = t_3 : k}{\Gamma \vdash t_1 = t_3 : k} \qquad \text{TY-TRANS}$$

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

$$\frac{\Gamma \vdash t = u : k \quad \Gamma \vdash \varphi : u}{\Gamma \vdash \varphi : t} \qquad \text{TY-SUBST}$$

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.

$$\frac{\Gamma \vdash \left\lceil t_i^1 = t_i^2 : \omega \right\rceil_1^m \quad \Gamma \vdash tv : \omega^m \to k_s}{\Gamma \vdash tv[t_1^1, \ldots, t_m^1] = tv[t_1^2, \ldots, t_m^2] : k_s} \qquad \text{TY-APPLY-1}$$

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

$$\frac{\Gamma \vdash u_1[\lceil t_i/\alpha_i \rceil_1^m] = u_2 : k_s \quad \Gamma \vdash tv[t_1, \ldots, t_m] : k_s}{(\Gamma', tv\colon k = \Lambda\alpha_1, \ldots, \alpha_m.u_1) \textbf{ as } \Gamma \vdash tv[t_1, \ldots, t_m] = u_2 : k_s} \qquad \text{TY-APPLY-2}$$

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 α_i. If the value is defined as $\textbf{ty\_pack}[u](v', u_0, \ldots, u_{m-1})$, then the type $v.i$ can be reduced to u_i.

$$\frac{i \in \{0 \ldots m-1\} \quad \Gamma \vdash u_i = t : \omega}{(\Gamma', v\colon \exists \alpha_0, \ldots, \alpha_{m-1}.t' = \textbf{ty\_pack}[u](v', u_0, \ldots, u_{m-1})) \textbf{ as } \Gamma \vdash v.i = t : \omega} \quad \text{TY-PROJECT-EQUAL}$$

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 $\Gamma \vdash \diamond$ is required in WF-TypeList since it can be the case that $n = 0$, i.e. the type lists are empty.

$$\frac{\Gamma \vdash \left\lceil t_i^1 = t_i^2 : \omega \right\rceil_1^n \quad \Gamma \vdash \diamond}{\Gamma \vdash \left\langle t_1^1, \ldots, t_n^1 \right\rangle = \left\langle t_1^2, \ldots, t_n^2 \right\rangle : \omega} \quad \text{WF-TYPELIST}$$

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.

$$\frac{\Gamma, \lceil \alpha_i\colon \omega \rceil_1^m \vdash \mathit{tydef}_s^1 = \mathit{tydef}_s^2 : k_s}{\Gamma \vdash \Lambda \alpha_1, \ldots, \alpha_m.\mathit{tydef}_s^1 = \Lambda \alpha_1, \ldots, \alpha_m.\mathit{tydef}_s^2 : \omega^m \to k_s} \quad \text{WF-TYDEF}$$

The next three rules allow progress to be made on proving the premise of WF-Tydef. They cover the cases in which tydef_s is a union or frame type[1]. All three rules require the context well-formedness premise $\Gamma \vdash \diamond$ since it can be the case that $n = 0$.

The union space $(\vec{t_1} + \cdots + \vec{t_n})$ is the disjoint union of several tuple[2] 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.

$$\frac{\Gamma \vdash \left\lceil \overrightarrow{t_i^1} = \overrightarrow{t_i^2} : \omega \right\rceil_1^n \quad \Gamma \vdash \diamond}{\Gamma \vdash \left(\overrightarrow{t_1^1} + \cdots + \overrightarrow{t_n^1}\right) = \left(\overrightarrow{t_1^2} + \cdots + \overrightarrow{t_n^2}\right) : \omega_{union[n]}} \quad \text{WF-TYDEFUNION}$$

Frames are doubly nested records. The frame type $\left\{l_1^f : R_1, \ldots, l_n^f : R_n\right\}$ declares sub-records named l_i^f with record type R_i, and the record type $\{l_1^r : t_1, \ldots, l_n^r : t_n\}$ declares fields named l_i^r 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.

$$\frac{\Gamma \vdash \left\lceil R_i^1 = R_i^2 : \textbf{record} \right\rceil_1^n \quad \Gamma \vdash \diamond}{\Gamma \vdash \left\{l_1^f : R_1^1, \ldots, l_n^f : R_n^1\right\} = \left\{l_1^f : R_1^2, \ldots, l_n^f : R_n^2\right\} : \omega_{\mathit{frame}}} \quad \text{WF-TYDEFFRAME}$$

$$\frac{\Gamma \vdash \left\lceil t_i^1 = t_i^2 : \Omega \right\rceil_1^n \quad \Gamma \vdash \diamond}{\Gamma \vdash \left\{l_1^r : t_1^1, \ldots, l_n^r : t_n^1\right\} = \left\{l_1^r : t_1^2, \ldots, l_n^r : t_n^2\right\} : \textbf{record}} \quad \text{WF-TYDEFRECORD}$$

[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 $\langle t_1, \ldots, t_n \rangle_{tuple\_class}$.

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.

$$\frac{\Gamma \vdash \lceil a_i : t \rceil_1^n \quad \Gamma \vdash t\ \mathbf{array} : \Omega}{\Gamma \vdash \langle a_1, \ldots, a_n \rangle : t\ \mathbf{array}} \quad \text{TY-STOREARRAY}$$

$$\frac{\Gamma \vdash \lceil a_i : t_i \rceil_1^n \quad \Gamma \vdash \langle t_1, \ldots, t_n \rangle_{tuple\_class} : \Omega}{\Gamma \vdash \langle a_1, \ldots, a_n \rangle : \langle t_1, \ldots, t_n \rangle_{tuple\_class}} \quad \text{TY-STORETUPLE}$$

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 $\mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\})$ 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 $\vec{t_1} + \cdots + \vec{t_n}$.
- The case $\vec{t_j}$ 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.

$$\frac{\begin{array}{l}\Gamma, \lceil \alpha_i : \omega = t'_i \rceil_1^m \vdash \lceil a_i : u_i \rceil_1^k \\ \Gamma \vdash \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) : \Omega \\ \Gamma \vdash tv[t'_1, \ldots, t'_m] = \vec{t_0} + \cdots + \vec{t}_{j-1} + \langle u_1, \ldots, u_k \rangle + \vec{t}_{j+1} + \cdots + \vec{t}_{n-1} : \omega_{union[n]}\end{array}}{\Gamma \vdash tv_j(a_1, \ldots, a_k) : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\})} \quad \text{TY-STOREUNION}$$

Functions are also polymorphic by default. Each function $\Lambda\alpha_1, \ldots, \alpha_m.\lambda v_1, \ldots, v_n.e$ has type parameters $\alpha_1, \ldots, \alpha_m$, actual parameters $v_1, \ldots, v_n$, and body e. The function has type $\forall \alpha_1, \ldots, \alpha_m.(u_1, \ldots, u_n) \to t$ if, given arbitrary types $\alpha_1, \ldots, \alpha_m$, and values $v_1 : u_1, \ldots, v_n : u_n$, the body e has type t.

$$\frac{\Gamma, \lceil v_i : u_i \rceil_1^n \vdash e : t}{\Gamma \vdash \lambda v_1, \ldots, v_n.e : (u_1, \ldots, u_n) \to t} \quad \text{TY-STOREFUN-MONO}$$

$$\frac{\Gamma, \lceil \alpha_i : \omega \rceil_1^m \vdash \lambda v_1, \ldots, v_n.e : (u_1, \ldots, u_n) \to t}{\Gamma \vdash \Lambda\alpha_1, \ldots, \alpha_m.\lambda v_1, \ldots, v_n.e : \forall \alpha_1, \ldots, \alpha_m.(u_1, \ldots, u_n) \to t} \quad \text{TY-STOREFUN-POLY}$$

Rawdata values are abstract. A value $\langle \mathrm{C} \rangle$ always has type **data**.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \langle \mathrm{C} \rangle : \mathbf{data}} \quad \text{TY-STOREDATA}$$

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

$$\frac{\Gamma \vdash tv[t_1, \ldots, t_m] : \omega_{frame}}{\Gamma \vdash \langle \mathrm{C} : \mathbf{frame}(tv[t_1, \ldots, t_m]) \rangle : \mathbf{frame}(tv[t_1, \ldots, t_m])} \quad \text{TY-STOREFRAME}$$

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 ω is used to denote types that are small. Since the WF-SMALLTYPE rule allows any ω type to be used as a normal type, equality judgments are given for the ω kind whenever possible.

$$\frac{\Gamma \vdash t_1 = t_2 : \omega}{\Gamma \vdash t_1 = t_2 : \Omega} \qquad \text{WF-SMALLTYPE}$$

Enumeration types are restricted to a maximum size of *max_enum*, which is a predefined small positive integer[3].

$$\frac{i \in \{0 \ldots max\_enum - 1\} \quad \Gamma \vdash \diamond}{\Gamma \vdash \mathbf{enum}_i = \mathbf{enum}_i : \omega} \qquad \text{WF-ST-TYENUM}$$

Type equality judgments for numeric FIR types are straightforward. Note that $\mathbb{Z}_{pre}^{sign}$ and $\mathbb{R}_{fpre}$ cannot be used as ω types.

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbb{Z}_{31} = \mathbb{Z}_{31} : \omega} \qquad \text{WF-ST-TYINT}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbb{Z}_{pre}^{sign} = \mathbb{Z}_{pre}^{sign} : \Omega} \qquad \text{WF-TYRAWINT}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbb{R}_{fpre} = \mathbb{R}_{fpre} : \Omega} \qquad \text{WF-TYFLOAT}$$

For the type $\mathbf{union}(tv[t_1, \ldots, t_m], set_I)$, set_I 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.

$$\frac{set_I \subseteq \{0 \ldots n-1\} \quad \Gamma \vdash \left\lceil t_i^1 = t_i^2 : \omega \right\rceil_1^m \quad \Gamma \vdash tv[t_1^2, \ldots, t_m^2] : \omega_{union[n]}}{\Gamma \vdash \mathbf{union}(tv[t_1^1, \ldots, t_m^1], set_I) = \mathbf{union}(tv[t_1^2, \ldots, t_m^2], set_I) : \omega} \qquad \text{WF-ST-TYUNION}$$

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

$$\frac{\Gamma \vdash \left\lceil t_i^1 = t_i^2 : \omega \right\rceil_1^n \quad \Gamma \vdash \diamond}{\Gamma \vdash \langle t_1^1, \ldots, t_n^1 \rangle_{\mathbf{normal}} = \langle t_1^2, \ldots, t_n^2 \rangle_{\mathbf{normal}} : \omega} \qquad \text{WF-ST-TYTUPLE-NORMAL}$$

The **box** tuples are used to "pack" non-ω types in a form that has kind ω. The tuple must have arity exactly one.

$$\frac{\Gamma \vdash t_1 = t_2 : \Omega}{\Gamma \vdash \langle t_1 \rangle_{\mathbf{box}} = \langle t_2 \rangle_{\mathbf{box}} : \omega} \qquad \text{WF-ST-TYTUPLE-BOXED}$$

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.

[3]Enumeration 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 *max_enum*. This reduces the likelihood that a dead block will be marked live by the garbage collector.

$$\frac{\Gamma \vdash t_1 = t_2 : \Omega}{\Gamma \vdash t_1 \ \mathbf{array} = t_2 \ \mathbf{array} : \omega} \qquad \text{WF-ST-TyArray}$$

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

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{data} = \mathbf{data} : \omega} \qquad \text{WF-ST-TyRawData}$$

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

$$\frac{\Gamma \vdash \left[t_i^1 = t_i^2 : \omega\right]_1^m \quad \Gamma \vdash tv[t_1^2, \ldots, t_m^2] : \omega_{frame}}{\Gamma \vdash \mathbf{frame}(tv[t_1^1, \ldots, t_m^1]) = \mathbf{frame}(tv[t_1^2, \ldots, t_m^2]) : \omega} \qquad \text{WF-ST-TyFrame}$$

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.

$$\frac{\Gamma, \left[\alpha_i : \omega\right]_1^m \vdash t_1 = t_2 : \Omega}{\Gamma \vdash \exists \alpha_1, \ldots, \alpha_m . t_1 = \exists \alpha_1, \ldots, \alpha_m . t_2 : \omega} \qquad \text{WF-ST-TyExists}$$

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 α_i.

$$\frac{i \in \{0 \ldots m-1\} \quad \Gamma \vdash v : \exists \alpha_0, \ldots, \alpha_{m-1}.t}{\Gamma \vdash v.i = v.i : \omega} \qquad \text{WF-ST-TyProject}$$

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.

$$\frac{\Gamma \vdash \left[t_i^1 = t_i^2 : \Omega\right]_1^n \quad \Gamma \vdash u^1 = u^2 : \Omega}{\Gamma \vdash ((t_1^1, \ldots, t_n^1) \to u^1) = ((t_1^2, \ldots, t_n^2) \to u^2) : \omega} \qquad \text{WF-ST-TyFun-Mono}$$

$$\frac{\Gamma, \left[\alpha_i : \omega\right]_1^m \vdash ((t_1^1, \ldots, t_n^1) \to u^1) = ((t_1^2, \ldots, t_n^2) \to u^2) : \omega}{\Gamma \vdash \forall \alpha_1, \ldots, \alpha_m.((t_1^1, \ldots, t_n^1) \to u^1) = \forall \alpha_1, \ldots, \alpha_m.((t_1^2, \ldots, t_n^2) \to u^2) : \omega} \qquad \text{WF-ST-TyFun-Poly}$$

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.

$$\frac{\Gamma, v : t \vdash \diamond}{\Gamma, v : t \vdash v : t} \qquad \text{Ty-AtomVar}$$

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

$$\frac{j \in \{0 \ldots i-1\} \quad \Gamma \vdash \mathbf{enum}_i : \Omega}{\Gamma \vdash \mathbf{enum}_i(j) : \mathbf{enum}_i} \quad \textsc{Ty-AtomEnum}$$

There are numeric constants for integers, raw integers (integers with various precision and signedness interpretations), and floating-point values. The side conditions $i \in \mathbb{Z}_{31}$, $r \in \mathbb{Z}_{pre}^{sign}$, and $x \in \mathbb{R}_{fpre}$ are necessary since syntax alone does not ensure that these values will be in the type.

$$\frac{i \in \mathbb{Z}_{31} \quad \Gamma \vdash \diamond}{\Gamma \vdash \mathbf{int}(i) : \mathbb{Z}_{31}} \quad \textsc{Ty-AtomInt}$$

$$\frac{r \in \mathbb{Z}_{pre}^{sign} \quad \Gamma \vdash \diamond}{\Gamma \vdash \mathbf{rawint}_{pre}^{sign}(r) : \mathbb{Z}_{pre}^{sign}} \quad \textsc{Ty-AtomRawInt}$$

$$\frac{x \in \mathbb{R}_{fpre} \quad \Gamma \vdash \diamond}{\Gamma \vdash \mathbf{float}_{fpre}(x) : \mathbb{R}_{fpre}} \quad \textsc{Ty-AtomFloat}$$

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.

$$\frac{r_{\mathit{off}} \in \mathbb{Z}_{32}^{\mathbf{signed}} \quad \Gamma \vdash tv[u_1, \ldots, u_m] = \left\{l^f : \{l^r : t; \ldots\}; \ldots\right\} : \omega_{frame}}{\Gamma \vdash \mathbf{label}(tv, l^f, l^r, r_{\mathit{off}}) : \mathbb{Z}_{32}^{\mathbf{signed}}} \quad \textsc{Ty-AtomLabel}$$

The sizeof construction has purpose similar to labels; the value $\mathbf{sizeof}(tv_1, \ldots, tv_n, r_{size})$ represents the sum of the (byte) sizes of the frames defined by $tv_1, \ldots, tv_n$ and the constant integer r_{size}. The size itself is determined by the runtime environment. The context well-formedness judgment is needed since n may be 0.

$$\frac{r_{size} \in \mathbb{Z}_{32}^{\mathbf{signed}} \quad \Gamma \vdash \lceil tv_i : \omega^{m_i} \to \omega_{frame} \rceil_1^n \quad \Gamma \vdash \diamond}{\Gamma \vdash \mathbf{sizeof}(tv_1, \ldots, tv_n, r_{size}) : \mathbb{Z}_{32}^{\mathbf{signed}}} \quad \textsc{Ty-AtomSizeof}$$

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

The constant atom is an optimization; zero-arity union cases can also be allocated using the allocation construction $\mathbf{union}(tv[t_1, \ldots, t_m], i)$ (and the corresponding type rule 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).

$$\frac{\begin{array}{l}\Gamma \vdash u = \mathbf{union}(tv[t_1, \ldots, t_m], \{i\}) : \omega \\ \Gamma \vdash tv[t_1, \ldots, t_m] = \vec{t}_0 + \cdots + \vec{t}_{i-1} + \langle\rangle + \vec{t}_{i+1} + \cdots + \vec{t}_{n-1} : \omega_{union[n]}\end{array}}{\Gamma \vdash \mathbf{const}[u](tv, i) : \mathbf{union}(tv[t_1, \ldots, t_m], \{i\})} \quad \textsc{Ty-AtomConst}$$

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 $\mathbf{ty\_apply}[u](a_f, u_1, \ldots, u_m)$ represents the function instantiated with type arguments $u_1, \ldots, u_m$.

$$\frac{\Gamma \vdash \lceil u_i : \omega \rceil_1^m \quad \Gamma \vdash a_f : \forall \alpha_1, \ldots, \alpha_m.t \quad \Gamma \vdash u = t[\lceil u_i/\alpha_i \rceil_1^m] : \omega}{\Gamma \vdash \mathbf{ty\_apply}[u](a_f, u_1, \ldots, u_m) : t[\lceil u_i/\alpha_i \rceil_1^m]} \quad \textsc{Ty-AtomTyApply}$$

A type *pack* construction is the existential introduction from. 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 α_i, then $\mathbf{ty\_pack}[\exists \alpha_1, \ldots, \alpha_m.t](v, t_1, \ldots, t_m)$ has type $\exists \alpha_1, \ldots, \alpha_m.t$.

$$\frac{\Gamma \vdash \lceil u_i : \omega \rceil_1^m \quad \Gamma \vdash v : t[\lceil u_i/\alpha_i \rceil_1^m] \quad \Gamma \vdash u = \exists \alpha_1, \ldots, \alpha_m.t : \omega}{\Gamma \vdash \mathbf{ty\_pack}[u](v, u_1, \ldots, u_m) : \exists \alpha_1, \ldots, \alpha_m.t} \quad \textsc{Ty-AtomTyPack}$$

The *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 $\mathbf{ty\_unpack}(v)$ has type t with the type projections $v.i$ substituted for α_i. The premise of Ty-AtomTyUnpack ensures that the type projections $v.i$ are well-formed.

$$\frac{\Gamma \vdash v : \exists \alpha_0, \ldots, \alpha_{m-1}.t}{\Gamma \vdash \mathbf{ty\_unpack}(v) : t[\lceil v.i/\alpha_i \rceil_0^{m-1}]} \quad \textsc{Ty-AtomTyUnpack}$$

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

$$\frac{\Gamma \vdash a : \mathbf{arg}(\mathit{unop}) \quad \Gamma \vdash \mathbf{res}(\mathit{unop}) : \Omega}{\Gamma \vdash \mathit{unop}\; a : \mathbf{res}(\mathit{unop})} \quad \textsc{Ty-AtomUnop}$$

The binary forms are similar. This rule uses the syntax tables defined in Section 6. The types $\mathbf{arg}_1(\mathit{binop})$ and $\mathbf{arg}_2(\mathit{binop})$ are the types of the argument to the binary operator *binop*, and $\mathbf{res}(\mathit{binop})$ is the type of the result.

$$\frac{\Gamma \vdash a_1 : \mathbf{arg}_1(\mathit{binop}) \quad \Gamma \vdash a_2 : \mathbf{arg}_2(\mathit{binop}) \quad \Gamma \vdash \mathbf{res}(\mathit{binop}) : \Omega}{\Gamma \vdash a_1\; \mathit{binop}\; a_2 : \mathbf{res}(\mathit{binop})} \quad \textsc{Ty-AtomBinop}$$

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 $\mathbf{let}\, v : t_1 = a \;\mathbf{in}\; e$ 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.

$$\frac{\Gamma \vdash a : t_1 \quad \Gamma, v : t_1 \vdash e : t_2}{\Gamma \vdash \mathbf{let}\, v : t_1 = a \;\mathbf{in}\; e : t_2} \quad \textsc{Ty-LetAtom}$$

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 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 $[\![\text{"s"}]\!]$ that defines a function that corresponds to the runtime operation.

$$\frac{\Gamma \vdash \lceil a_i : u_i \rceil_1^n \quad \Gamma, v : t_1 \vdash e : t_2 \quad \Gamma \vdash [\![\text{"s"}]\!] : (u_1, \ldots, u_n) \to t_1}{\Gamma \vdash \textbf{let}\, v : t_1 = (\text{"}s\text{"} : (u_1, \ldots, u_n) \to t_1)(a_1, \ldots, a_n) \textbf{ in } e : t_2} \quad \textsc{Ty-LetExt}$$

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 $t_1, \ldots, t_m$.

$$\frac{\Gamma \vdash a_f : (u_1, \ldots, u_n) \to \textbf{enum}_0 \quad \Gamma \vdash \lceil a_i : u_i \rceil_1^n}{\Gamma \vdash a_f(a_1, \ldots, a_n) : \textbf{enum}_0} \quad \textsc{Ty-TailCall}$$

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-Special-Call uses a common form for all of the special-calls. If *tailop* is a special-call, then **special** *tailop* is an expression.

$$\frac{\Gamma \vdash \mathit{tailop} : \textbf{special}\, t}{\Gamma \vdash \textbf{special}\, \mathit{tailop} : t} \quad \textsc{Ty-Special-Call}$$

Process migration is specified with the special-call expression **migrate** $[j, a_{ptr}, a_{off}]\ a_f(a_1, \ldots, a_n)$, which is described in Section 8.2. The atoms a_{ptr} and a_{off} specify a string (as a rawdata 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.

$$\frac{\begin{array}{cc} & \Gamma \vdash a_{ptr} : \textbf{data} \\ j \in \mathbb{Z}_{31} & \Gamma \vdash a_{off} : \mathbb{Z}_{32}^{\textbf{signed}} \\ \Gamma \vdash \lceil a_i : t_i \rceil_1^n & \Gamma \vdash a_f : (t_1, \ldots, t_n) \to \textbf{enum}_0 \end{array}}{\Gamma \vdash \textbf{migrate}\ [j, a_{ptr}, a_{off}]\ a_f(a_1, \ldots, a_n) : \textbf{special}\, \textbf{enum}_0} \quad \textsc{Ty-SysMigrate}$$

The "atomic" special-calls are described in Section 8.3. The expression **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 $\mathbb{Z}_{32}^{\textbf{signed}}$.

$$\frac{\Gamma \vdash a_{const} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma \vdash \lceil a_i : t_i \rceil_1^n \quad \Gamma \vdash a_f : (\mathbb{Z}_{32}^{\textbf{signed}}, t_1, \ldots, t_n) \to \textbf{enum}_0}{\Gamma \vdash \textbf{atomic}\ a_f(a_{const}, a_1, \ldots, a_n) : \textbf{special}\, \textbf{enum}_0} \quad \textsc{Ty-Atomic}$$

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** $a'_{fun}(a'_{const}, a'_1, \ldots, a'_n)$, the **rollback** $[a_{level}, a_{const}]$ special-call expression resumes execution with a tail-call to $a'_{fun}(a_{const}, a'_1, \ldots, a'_n)$.

$$\frac{\Gamma \vdash a_{level} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma \vdash a_{const} : \mathbb{Z}_{32}^{\textbf{signed}}}{\Gamma \vdash \textbf{rollback}\ [a_{level}, a_{const}] : \textbf{special}\,\textbf{enum}_0} \qquad \textsc{Ty-AtomicRollback}$$

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

$$\frac{\Gamma \vdash a_{level} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma \vdash \lceil a_i : t_i \rceil_1^n \quad \Gamma \vdash a_f : (t_1, \ldots, t_n) \to \textbf{enum}_0}{\Gamma \vdash \textbf{commit}\ [a_{level}]\ a_f(a_1, \ldots, a_n) : \textbf{special}\,\textbf{enum}_0} \qquad \textsc{Ty-AtomicCommit}$$

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 $\textbf{match}\,a\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n$ 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.

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{31} \quad \Gamma \vdash a : \mathbb{Z}_{31} \quad \Gamma \vdash \lceil e_i : t \rceil_1^n}{\Gamma \vdash \textbf{match}\,a\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t} \qquad \textsc{Ty-Match-Int}$$

$$\frac{j > 0 \quad \lceil s_i \in set_I \rceil_1^n \quad \bigcup_{i=1}^n s_i = \textbf{enum}_j \quad \Gamma \vdash a : \textbf{enum}_j \quad \Gamma \vdash \lceil e_i : t \rceil_1^n}{\Gamma \vdash \textbf{match}\,a\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t} \qquad \textsc{Ty-Match-Enum}$$

$$\frac{\lceil s_i \in set_R \rceil_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{pre}^{sign} \quad \Gamma \vdash a : \mathbb{Z}_{pre}^{sign} \quad \Gamma \vdash \lceil e_i : t \rceil_1^n}{\Gamma \vdash \textbf{match}\,a\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t} \qquad \textsc{Ty-Match-RawInt}$$

The match case for unions is a special case. A value v of type $\textbf{union}(tv[t_1, \ldots, t_m], s_{union})$ belongs to some union space $\vec{t_1} + \cdots + \vec{t_n}$. The actual value for v belongs to exactly one of the cases in the union, and the $\textbf{match}\,v\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n$ 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_{union}. The match cases s_i must include all of the possible values in s_{union}. For case $s_j \mapsto e_j$, the expression e_j must be well-typed given the *restricted* type $\textbf{union}(tv[t_1, \ldots, t_m], s_j)$ for v.

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \{\} \neq s_{union} = \bigcup_{i=1}^n s_i \quad \lceil \Gamma, v\colon \textbf{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t \rceil_1^n}{\Gamma, v\colon \textbf{union}(tv[t_1, \ldots, t_m], s_{union}) \vdash \textbf{match}\,v\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t} \qquad \textsc{Ty-Match-Union}$$

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 $\mathbb{Z}_{32}^{\textbf{signed}}$; this alternation is expressed with the **offset** macro. Other operations restrict the offset atom to be a constant; these use the $\textbf{offset}_c^{\textbf{const}}(i)$ macro, where c is a *tuple_class* and i is a word offset into a tuple. Note that *word_size* is a constant, and indicates the size of a word on the architecture.

$$\begin{array}{rcl} \textbf{offset} & = & \mathbb{Z}_{31} \mid \mathbb{Z}_{32}^{\textbf{signed}} \\ \textbf{offset}_c^{\textbf{const}}(i) & = & \begin{cases} \textbf{int}(i) \mid \textbf{rawint}_{32}^{\textbf{signed}}(i \cdot \mathit{word\_size}) & \text{when } c = \textbf{normal} \\ \textbf{int}(0) \mid \textbf{rawint}_{32}^{\textbf{signed}}(0) & \text{when } c = \textbf{box} \end{cases} \end{array}$$

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

$$\frac{\Gamma \vdash \mathit{alloc} : \textbf{alloc}\, u \quad \Gamma, v : u \vdash e : t}{\Gamma \vdash \textbf{let}\, v = \mathit{alloc}\ \textbf{in}\ e : t} \quad \textsc{Ty-Alloc}$$

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.

$$\frac{\Gamma \vdash \lceil a_i : u_i \rceil_1^n \quad \Gamma \vdash \langle u_1, \ldots, u_n \rangle_c : \omega}{\Gamma \vdash \langle a_1, \ldots, a_n \rangle_c : \textbf{alloc}\, \langle u_1, \ldots, u_n \rangle_c} \quad \textsc{Ty-Alloc-Tuple}$$

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

$$\frac{\Gamma \vdash a_{size} : \textbf{offset} \quad \Gamma \vdash a_{init} : u}{\Gamma \vdash \textbf{array}(a_{size}, a_{init}) : \textbf{alloc}\, u\ \textbf{array}} \quad \textsc{Ty-Alloc-Array}$$

Unions are allocated with the form $\textbf{union}(tv[a_1, \ldots, a_k], j)$, 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.

$$\frac{\Gamma \vdash tv_j(a_1, \ldots, a_k) : \textbf{union}(tv[t_1, \ldots, t_m], \{j\})}{\Gamma \vdash \textbf{union}(tv[a_1, \ldots, a_k], j) : \textbf{alloc}\, \textbf{union}(tv[t_1, \ldots, t_m], \{j\})} \quad \textsc{Ty-Alloc-Union}$$

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

$$\frac{\Gamma \vdash a : \textbf{offset}}{\Gamma \vdash \textbf{malloc}(a) : \textbf{alloc}\, \textbf{data}} \quad \textsc{Ty-Alloc-Malloc}$$

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

[6] For readability, we shorten $\langle a_1, \ldots, a_n \rangle_c : t$ to $\langle a_1, \ldots, a_n \rangle_c$ 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 $\mathbf{frame}(tv[u_1, \ldots, u_m])$ is well-formed. The number of bytes allocated is at least $\mathbf{sizeof}(tv, 1)$.

$$\frac{\Gamma \vdash \mathbf{frame}(tv[u_1, \ldots, u_m]) : \omega}{\Gamma \vdash \mathbf{frame}(tv[u_1, \ldots, u_m]) : \mathbf{alloc}\,\mathbf{frame}(tv[u_1, \ldots, u_m])} \quad \textsc{Ty-Alloc-Frame}$$

4.8.5 Subscripting typing rules

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

In general, $\mathbf{let}\, v \colon t_1 = a_1[a_2]\ \mathbf{in}\ e$ 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, $a_1[a_2] \colon t_1 \leftarrow a_3;\ e$ 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 $\langle u_0, \ldots, u_{n-1} \rangle_c$, the element has type u_j.

$$\frac{\Gamma \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \Gamma, v_1 : u \vdash e : t}{\Gamma \vdash \mathbf{let}\, v_1 : u = a_1[\mathbf{offset}_c^{\mathbf{const}}(j)]\ \mathbf{in}\ e : t} \quad \textsc{Ty-LetSub-Tuple}$$

$$\frac{\Gamma \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \Gamma \vdash a_3 : u \quad \Gamma \vdash e : t}{\Gamma \vdash a_1[\mathbf{offset}_c^{\mathbf{const}}(j)] \colon u \leftarrow a_3;\ e : t} \quad \textsc{Ty-SetSub-Tuple}$$

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 **array**, and the element being accessed has type u.

$$\frac{\Gamma \vdash a_1 : u\ \mathbf{array} \quad \Gamma \vdash a_2 : \mathbf{offset} \quad \Gamma, v : u \vdash e : t}{\Gamma \vdash \mathbf{let}\, v \colon u = a_1[a_2]\ \mathbf{in}\ e : t} \quad \textsc{Ty-LetSub-Array}$$

$$\frac{\Gamma \vdash a_1 : u\ \mathbf{array} \quad \Gamma \vdash a_2 : \mathbf{offset} \quad \Gamma \vdash a_3 : u \quad \Gamma \vdash e : t}{\Gamma \vdash a_1[a_2] \colon u \leftarrow a_3;\ e : t} \quad \textsc{Ty-SetSub-Array}$$

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 $\vec{u}_j$ in some union type $\vec{u}_0 + \cdots + \vec{u}_{n-1}$. The index a_2 must be a constant k. The type list $\vec{u}_j$ must be a vector $\langle u'_0, \ldots, u'_{l-1} \rangle$, and the element being accessed must have type u'_k.

$$\frac{\begin{array}{l}\Gamma \vdash a_1 : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) \\ \Gamma \vdash tv[t'_1, \ldots, t'_m] = \left(\lceil \vec{u}_i \rceil_0^{j-1} + \left\langle \lceil u'_i \rceil_0^{k-1}, u, \lceil u'_i \rceil_{k+1}^{l-1} \right\rangle + \lceil \vec{u}_i \rceil_{j+1}^{n-1} \right) : \omega_{union[n]} \\ \Gamma, v\colon u \vdash e : t \end{array}}{\Gamma \vdash \mathbf{let}\, v\colon u = a_1[\mathbf{offset}^{\mathbf{const}}_{\mathbf{normal}}(k)]\ \mathbf{in}\ e : t} \quad \text{TY-LETSUB-UNION}$$

$$\frac{\begin{array}{l}\Gamma \vdash a_1 : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) \\ \Gamma \vdash tv[t'_1, \ldots, t'_m] = \left(\lceil \vec{u}_i \rceil_0^{j-1} + \left\langle \lceil u'_i \rceil_0^{k-1}, u, \lceil u'_i \rceil_{k+1}^{l-1} \right\rangle + \lceil \vec{u}_i \rceil_{j+1}^{n-1} \right) : \omega_{union[n]} \\ \Gamma \vdash a_3 : u \\ \Gamma \vdash e : t \end{array}}{\Gamma \vdash a_1[\mathbf{offset}^{\mathbf{const}}_{\mathbf{normal}}(k)]\colon u \leftarrow a_3;\ e : t} \quad \text{TY-SETSUB-UNION}$$

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.

$$\frac{\Gamma \vdash a_1 : \mathbf{data} \quad \Gamma \vdash a_2 : \mathbf{offset} \quad \Gamma, v\colon u \vdash e : t}{\Gamma \vdash \mathbf{let}\, v\colon u = a_1[a_2]\ \mathbf{in}\ e : t} \quad \text{TY-LETSUB-RAWDATA}$$

$$\frac{\Gamma \vdash a_1 : \mathbf{data} \quad \Gamma \vdash a_2 : \mathbf{offset} \quad \Gamma \vdash a_3 : u \quad \Gamma \vdash e : t}{\Gamma \vdash a_1[a_2]\colon u \leftarrow a_3;\ e : t} \quad \text{TY-SETSUB-RAWDATA}$$

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 $\langle l^f, l^r \rangle$, a_1 is a frame, and the element being accessed has the same type as the field $a_1.l^f.l^r$.

$$\frac{\begin{array}{l} r_{\mathit{off}} \in \mathbb{Z}_{32}^{\mathbf{signed}} \\ \Gamma \vdash a_1 : \mathbf{frame}(tv[t'_1, \ldots, t'_m]) \\ \Gamma \vdash tv[t'_1, \ldots, t'_m] = \left\{ l^f : \{ l^r : u; \cdots \} ; \cdots \right\} : \omega_{frame} \\ \Gamma, v\colon u \vdash e : t \end{array}}{\Gamma \vdash \mathbf{let}\, v\colon u = a_1[\mathbf{label}(tv, l^f, l^r, r_{\mathit{off}})]\ \mathbf{in}\ e : t} \quad \text{TY-LETSUB-FRAME}$$

$$\frac{\begin{array}{l} r_{\mathit{off}} \in \mathbb{Z}_{32}^{\mathbf{signed}} \\ \Gamma \vdash a_1 : \mathbf{frame}(tv[t_1, \ldots, t_m]) \\ \Gamma \vdash tv[t_1, \ldots, t_m] = \left\{ l^f : \{ l^r : u; \cdots \} ; \cdots \right\} : \omega_{frame} \\ \Gamma \vdash a_3 : u \\ \Gamma \vdash e : t \end{array}}{\Gamma \vdash a_1[\mathbf{label}(tv, l^f, l^r, r_{\mathit{off}})]\colon u \leftarrow a_3;\ e : t} \quad \text{TY-SETSUB-FRAME}$$

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.

$$\frac{\Gamma, l\colon t \vdash \diamond}{\Gamma, l\colon t \vdash l : t} \qquad \textsc{Ty-Label}$$

Labels are *not* variables; there is no atom that allows global values to be used directly. The value in a global is loaded with the $\mathbf{let}\, v\colon t_1 = \mathbf{global}\, l\ \mathbf{in}\ e$ expression, and stored with the $\mathbf{global}\, l\colon t_1 \leftarrow a;\ e$ expression.

$$\frac{\Gamma \vdash l : u \quad \Gamma, v\colon u \vdash e : t}{\Gamma \vdash \mathbf{let}\, v\colon u = \mathbf{global}\, l\ \mathbf{in}\ e : t} \qquad \textsc{Ty-LetGlobal}$$

$$\frac{\Gamma \vdash l : u \quad \Gamma \vdash a : u \quad \Gamma \vdash e : t}{\Gamma \vdash \mathbf{global}\, l\colon u \leftarrow a;\ e : t} \qquad \textsc{Ty-SetGlobal}$$

Chapter 5

Operational Semantics

Evaluation is defined on *programs*, which include three parts: the current environment Γ, a checkpoint environment $\mathcal{C}$, 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 $\langle \Gamma, f(\diamond, a_1, \ldots, a_n)\rangle$ contains a context Γ, and a function $f(\diamond, a_1, \ldots, a_n)$, where $\diamond$ is a special transaction parameter. The function is called if evaluation is resumed from the checkpoint.

$$
\begin{array}{lcll}
C & ::= & \langle \Gamma, f(\diamond, a_1, \ldots, a_n)\rangle & \text{Single checkpoint} \\
\mathcal{C} & ::= & C_m; \ldots; C_1 & \text{Checkpoint environment}
\end{array}
$$

Definition 5.0.1 Fully-defined contexts

A context Γ 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 $tv: k = tydef$, 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 $(\Gamma \mid \mathcal{C} \mid e)$ that satisfies the following conditions.*

- Γ *is fully-defined and* $\Gamma \vdash e : \mathbf{enum}_0$,
- *For each $\langle \Gamma', f(\diamond, a_1, \ldots, a_n)\rangle \in \mathcal{C}$, the context Γ' is fully-defined, and the judgment $\Gamma', v_{const}: \mathbb{Z}_{32}^{\mathbf{signed}} \vdash f(v_{const}, a_1, \ldots, a_n) : \mathbf{enum}_0$ 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

$$(\Gamma \mid \mathcal{C} \mid e) \rightarrow (\Gamma' \mid \mathcal{C}' \mid e')$$

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

$$(\Gamma \mid \mathcal{C} \mid e) \rightarrow \mathbf{error}$$

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 $\mathbf{int}(i)$ or $\mathbf{rawint}^{sign}_{pre}(r)$. 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" [], 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 $[\![op]\!]$ 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 $[\![+_{\mathbb{Z}_{31}}]\!]$ for the $\mathbb{Z}_{31}$ addition operator is actual 31 bit, signed addition. Note that in RED-ATOM-UNOP, we use informal syntax; i and j are taken to be atoma with appropriate values for the operator being evaluated. Similar informal syntax is used in RED-ATOM-BINOP[1].

$$(\Gamma \mid \mathcal{C} \mid E[\mathit{unop}\, i]) \to (\Gamma \mid \mathcal{C} \mid E[[\![\mathit{unop}]\!](i)]) \qquad \text{RED-ATOM-UNOP}$$

$$(\Gamma \mid \mathcal{C} \mid E[i\ \mathit{binop}\ j]) \to (\Gamma \mid \mathcal{C} \mid E[[\![\mathit{binop}]\!](i,j)]) \qquad \text{RED-ATOM-BINOP}$$

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

$$(\Gamma, v\colon t = h \mid \mathcal{C} \mid E[v]) \to (\Gamma, v\colon t = h \mid \mathcal{C} \mid E[h]) \qquad \text{RED-ATOM-VAR}$$

Type application is used to instantiate polymorphic functions. The type parameters of the function $f = \Lambda\alpha_1, \ldots, \alpha_m.b$ 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.

$$\begin{array}{l}((\Gamma', f\colon u' = \Lambda\alpha_1, \ldots, \alpha_m.b) \textbf{ as } \Gamma \mid \mathcal{C} \mid E[\mathbf{ty\_apply}[u](f, u_1, \ldots, u_m)]) \to \\ \quad (\Gamma, \lceil \alpha_i : \omega = u_i \rceil_1^m, g\colon u = b \mid \mathcal{C} \mid E[g])\end{array} \qquad \text{RED-ATOM-APPLY}$$

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

[1]It would be straightforward to list out a reduction rule for each unary and binary operator.

$$
\begin{array}{rcl}
A & ::= & [] \\
 & | & unop\ A \\
 & | & A\ binop\ a \\
 & | & h\ binop\ A \\
 \\
S & ::= & \textbf{migrate}\ [j, A, a_o]\ a_{fun}(a_1, \ldots, a_n) \\
 & | & \textbf{migrate}\ [j, h_p, A]\ a_{fun}(a_1, \ldots, a_n) \\
 & | & \textbf{migrate}\ [j, h_p, h_o]\ A(a_1, \ldots, a_n) \\
 & | & \textbf{migrate}\ [j, h_p, h_o]\ h_{fun}(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \\
 & | & \textbf{atomic}\ A(a_{const}, a_1, \ldots, a_n) \\
 & | & \textbf{atomic}\ v(A, a_1, \ldots, a_n) \\
 & | & \textbf{atomic}\ v(h_{const}, h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \\
 & | & \textbf{rollback}\ [A, a_{const}] \\
 & | & \textbf{rollback}\ [h_{level}, A] \\
 & | & \textbf{commit}\ [A]\ a_{fun}(a_1, \ldots, a_n) \\
 & | & \textbf{commit}\ [h_{level}]\ A(a_1, \ldots, a_n) \\
 & | & \textbf{commit}\ [h_{level}]\ v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \\
 \\
L & ::= & \langle h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n \rangle_{tuple\_class} : t \\
 & | & \textbf{union}(tv[h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n], i) : t \\
 & | & \textbf{array}(A, a_{init}) : t \\
 & | & \textbf{array}(h_{size}, A) : t \\
 & | & \textbf{malloc}(A) : t \\
 \\
E & ::= & \textbf{let}\ v : t = A\ \textbf{in}\ e \\
 & | & \textbf{let}\ v : t = (\text{“}s\text{”} : t_s)(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n)\ \textbf{in}\ e \\
 & | & A(a_1, \ldots, a_n) \\
 & | & v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \\
 & | & \textbf{special}\ S \\
 & | & \textbf{match}\ A\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n \\
 & | & \textbf{let}\ v = L\ \textbf{in}\ e \\
 & | & \textbf{let}\ v : t = A[a_2]\ \textbf{in}\ e \\
 & | & \textbf{let}\ v : t = h[A]\ \textbf{in}\ e \\
 & | & A[a_2] : t \leftarrow a_3;\ e \\
 & | & v[A] : t \leftarrow a_3;\ e \\
 & | & v[h] : t \leftarrow A;\ e \\
 & | & \textbf{global}\ l : t \leftarrow A;\ e
\end{array}
$$

Figure 5.1: Evaluation contexts

$$
\begin{array}{l}
((\Gamma', v_1 : u' = b)\ \textbf{as}\ \Gamma \mid \mathcal{C} \mid E[\textbf{ty\_pack}[u](v_1, u_0, \ldots, u_{m-1})]) \rightarrow \\
\quad (\Gamma, v_2 : u = \textbf{ty\_pack}[u](v_1, u_0, \ldots, u_{m-1}) \mid \mathcal{C} \mid E[v_2])
\end{array}
\qquad \text{RED-ATOM-PACK}
$$

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 $\alpha_0, \ldots, \alpha_{m-1}$.

$$
\begin{array}{l}
\Big((\Gamma', v_1 : u = \textbf{ty\_pack}[\exists \lceil \alpha_i \rceil_0^{m-1}.t](v, \lceil u_i \rceil_0^{m-1}))\ \textbf{as}\ \Gamma \mid \mathcal{C} \mid E[\textbf{ty\_unpack}(v_1)]\Big) \rightarrow \\
\quad \Big(\Gamma, \lceil \alpha_i : \omega = u_i \rceil_0^{m-1}, v_2 : t = v \mid \mathcal{C} \mid E[v_2]\Big)
\end{array}
\qquad \text{RED-ATOM-UNPACK}
$$

5.3 Basic expressions

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

Evaluation of **let** $v\colon u = h$ **in** e adds the definition $v\colon u = h$ to the context, then evaluates the expression e.

$$(\Gamma \mid \mathcal{C} \mid \mathbf{let}\, v\colon u = h \;\mathbf{in}\; e) \rightarrow (\Gamma, v\colon u = h \mid \mathcal{C} \mid e) \qquad \text{RED-LETATOM}$$

For external calls, we defer to a semantic interpretation function $[\![\text{“s”}]\!]$ that specifies the interpretation of the built-in function named “s”.

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \mathbf{let}\, v\colon t = (\text{“}s\text{”} : (u_1, \ldots, u_n) \rightarrow t)(h_1, \ldots, h_n) \;\mathbf{in}\; e) \rightarrow \\ \quad (\Gamma, v\colon t = [\![\text{“s”}]\!](h_1, \ldots, h_n) \mid \mathcal{C} \mid e)\end{array} \qquad \text{RED-LETEXT}$$

A tail-call to function $(\lambda v_1, \ldots, v_n.e)$ with arguments $(h_1, \ldots, h_n)$ adds the definitions $\lceil v_i\colon t_i = h_i\rceil_1^n$ to the evaluation context, and evaluates the expression e.

$$\begin{array}{l}((\Gamma', v\colon (u_1, \ldots, u_n) \rightarrow t = \lambda v_1, \ldots, v_n.e) \;\mathbf{as}\; \Gamma \mid \mathcal{C} \mid v(h_1, \ldots, h_n)) \rightarrow \\ \quad (\Gamma, \lceil v_i\colon u_i = h_i\rceil_1^n \mid \mathcal{C} \mid e)\end{array} \qquad \text{RED-TAILCALL}$$

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 Γ, $\mathcal{C}$, 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 **migrate** $[j, a_{ptr}, a_{off}]$ $f(a_1, \ldots, a_n)$ ignores the location described in the string $a_{ptr}[a_{off}]$, and acts as a tail-call $f(a_1, \ldots, a_n)$.

$$(\Gamma \mid \mathcal{C} \mid \mathbf{special\,migrate}\; [j, h_{ptr}, h_{off}]\; f(h_1, \ldots, h_n)) \rightarrow (\Gamma \mid \mathcal{C} \mid f(h_1, \ldots, h_n)) \qquad \text{RED-SYSMIGRATE}$$

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

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \mathbf{special\,atomic}\; f(h_{const}, h_1, \ldots, h_n)) \rightarrow \\ \quad (\Gamma \mid \langle \Gamma, f(\diamond, h_1, \ldots, h_n)\rangle \,; \mathcal{C} \mid f(h_{const}, h_1, \ldots, h_n))\end{array} \qquad \text{RED-ATOMIC}$$

When a transaction with checkpoint $\langle \Gamma, f(\diamond, a_1, \ldots, a_n)\rangle$ is rolled back with the **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 Γ and the truncated checkpoint environment $C_i; \ldots; C_1$. All checkpoints with level higher than i are discarded[2].

[2] The 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.

$$\begin{array}{l}(\Gamma' \mid C_m;\ldots;C_i = \langle \Gamma, f(\diamond, h_1,\ldots,h_n)\rangle;\ldots;C_1 \mid \textbf{special}\,\textbf{rollback}\ [i,j]) \rightarrow \\ \quad (\Gamma \mid C_i;C_{i-1};\ldots;C_1 \mid f(j,h_1,\ldots,h_n)) \\ \quad \textbf{when}\, i \in \{1\ldots m\}\end{array} \qquad \textsc{Red-Atomic-Rollback-1}$$

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.

$$\begin{array}{l}(\Gamma' \mid C_m = \langle \Gamma, f(\diamond, h_1,\ldots,h_n)\rangle;\ldots;C_1 \mid \textbf{special}\,\textbf{rollback}\ [0,j]) \rightarrow \\ \quad (\Gamma \mid C_m;\ldots;C_1 \mid f(j,h_1,\ldots,h_n))\end{array} \qquad \textsc{Red-Atomic-Rollback-2}$$

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

$$\begin{array}{l}(\Gamma \mid C_m;\ldots;C_1 \mid \textbf{special}\,\textbf{rollback}\ [i,j]) \rightarrow \textbf{error} \\ \quad \textbf{when}\, i \notin \{0\ldots m\} \vee m = 0\end{array} \qquad \textsc{Red-Atomic-Rollback-Error}$$

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 $f(a_1,\ldots,a_n)$.

$$\begin{array}{l}(\Gamma \mid C_m;\ldots;C_1 \mid \textbf{special}\,\textbf{commit}\ [i]\ f(h_1,\ldots,h_n)) \rightarrow \\ \quad (\Gamma \mid C_m;\ldots;C_{i+1};C_{i-1};\ldots;C_1 \mid f(h_1,\ldots,h_n)) \\ \quad \textbf{when}\, i \in \{1\ldots m\}\end{array} \qquad \textsc{Red-Atomic-Commit-1}$$

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.

$$\begin{array}{l}(\Gamma \mid C_m;\ldots;C_1 \mid \textbf{special}\,\textbf{commit}\ [0]\ f(h_1,\ldots,h_n)) \rightarrow \\ \quad (\Gamma \mid C_{m-1};\ldots;C_1 \mid f(h_1,\ldots,h_n))\end{array} \qquad \textsc{Red-Atomic-Commit-2}$$

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

$$\begin{array}{l}(\Gamma \mid C_m;\ldots;C_1 \mid \textbf{special}\,\textbf{commit}\ [i]\ f(h_1,\ldots,h_n)) \rightarrow \textbf{error} \\ \quad \textbf{when}\, i \notin \{0\ldots m\} \vee m = 0\end{array} \qquad \textsc{Red-Atomic-Commit-Error}$$

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.

$$(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v = \langle h_1,\ldots,h_n\rangle_c : u\ \textbf{in}\ e) \rightarrow (\Gamma, v\colon u = \langle h_1,\ldots,h_n\rangle \mid \mathcal{C} \mid e) \qquad \textsc{Red-Alloc-Tuple}$$

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

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v = \textbf{array}(i_{size}, h_{in}) : t\ \textbf{in}\ e) \rightarrow \left(\Gamma, v\colon t = \left\langle h_{in}^1,\ldots,h_{in}^{i_{size}}\right\rangle \mid \mathcal{C} \mid e\right) \\ \quad \textbf{when}\, i_{size} \geq 0\end{array} \qquad \textsc{Red-Alloc-Array}$$

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

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v = \textbf{array}(i_{size}, h_{in}) : t\ \textbf{in}\ e) \rightarrow \textbf{error} \\ \quad \textbf{when}\, i_{size} < 0\end{array} \qquad \textsc{Red-Alloc-Array-Error}$$

Union constructor allocation adds the value to the environment.

$$(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v = \textbf{union}(tv[h_1, \ldots, h_n], j) : t \textbf{ in } e) \rightarrow (\Gamma, v \colon t = tv_j(h_1, \ldots, h_n) \mid \mathcal{C} \mid e) \qquad \textsc{Red-Alloc-Union}$$

We do not give precise semantics of rawdata aggregates. The heap value is represented with the term $\langle \textsc{c} \rangle$, 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.

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v = \textbf{malloc}(i_{size}) : t \textbf{ in } e) \rightarrow (\Gamma, v \colon t = \langle \textsc{c} \rangle \mid \mathcal{C} \mid e) \\ \quad \textbf{when}\, i_{size} \geq 0\end{array} \qquad \textsc{Red-Alloc-Malloc}$$

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

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v = \textbf{malloc}(i_{size}) : t \textbf{ in } e) \rightarrow \textbf{error} \\ \quad \textbf{when}\, i_{size} < 0\end{array} \qquad \textsc{Red-Alloc-Malloc-Error}$$

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

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v = \textbf{frame}(tv[t_1, \ldots, t_m]) \textbf{ in } e) \rightarrow \\ \quad (\Gamma, v \colon \textbf{frame}(tv[t_1, \ldots, t_m]) = \langle \textsc{c} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \mid \mathcal{C} \mid e)\end{array} \qquad \textsc{Red-Alloc-Frame}$$

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

$$(i \in s_k) \wedge \forall j \in \{1, \ldots, k-1\}.(i \notin s_j)$$

forces the choice of the first possible match.

$$\begin{array}{l}\left(\Gamma \mid \mathcal{C} \mid \textbf{match}\, i \textbf{ with } \lceil s_j \mapsto e_j \rceil_{j=1}^{n}\right) \rightarrow (\Gamma \mid \mathcal{C} \mid e_k) \\ \quad \textbf{when } (i \in s_k) \wedge \forall j \in \{1, \ldots, k-1\}.(i \notin s_j)\end{array} \qquad \textsc{Red-Match-Int}$$

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 Ty-Match-Union, the type of the value being matched is adjusted to reflect the more specific tag.

$$\begin{array}{l}\left(\Gamma, v \colon \textbf{union}(tv[t_1, \ldots, t_m], s) = tv_j(a_1, \ldots, a_n) \mid \mathcal{C} \mid \textbf{match}\, v \textbf{ with } \lceil s_i \mapsto e_i \rceil_1^l\right) \rightarrow \\ \quad (\Gamma, v \colon \textbf{union}(tv[t_1, \ldots, t_m], s_k) = tv_j(a_1, \ldots, a_n) \mid \mathcal{C} \mid e_k) \\ \quad \textbf{when } (j \in s_k) \wedge \forall p \in \{1, \ldots, k-1\}.(i \notin s_p)\end{array} \qquad \textsc{Red-Match-Union}$$

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.

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

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

Arrays are similar to tuples when the array index is in bounds.

$$((\Gamma', v_2 : u' = \langle h_0, \ldots, h_{n-1} \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow \\ (\Gamma, v_1 : u = h_j \mid \mathcal{C} \mid e) \\ \textbf{when}\, j \in \{0 \ldots n-1\} \qquad \textsc{Red-LetSub-Array}$$

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

It is an error for an array subscript to be out-of-bounds.

$$((\Gamma', v_2 : t_2 = \langle h_0, \ldots, h_{n-1} \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : t_1 = v_2[j] \textbf{ in } e) \rightarrow \\ \textbf{error} \\ \textbf{when}\, j \notin \{0 \ldots n-1\} \qquad \textsc{Red-LetSub-Array-Error}$$

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

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.

$$((\Gamma', v_2 : u' = tv_j(h_0, \ldots, h_{n-1})) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[k] \textbf{ in } e) \rightarrow \\ (\Gamma, v_1 : u = h_k \mid \mathcal{C} \mid e) \qquad \textsc{Red-LetSub-Union}$$

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

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 $\textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : t)$ that projects a valid value h of type t from data area $\langle \textsc{c} \rangle$, or results in an error. We also assume the existence of a runtime function $\textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : t \leftarrow h)$ to store a value in a rawdata aggregate, returning a new value $\langle \textsc{c}' \rangle$ or producing an error.

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

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

Given these runtime functions, a rawdata subscript operation returns the value given by the runtime, or produces an error.

$$((\Gamma', v_2 \colon u' = \langle \mathrm{C} \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \to \\ (\Gamma, v_1 : u = h \mid \mathcal{C} \mid e) \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} \rangle \, [j] : u) = h \qquad \text{RED-LETSUB-RAWDATA}$$

$$((\Gamma', v_2 \colon u' = \langle \mathrm{C} \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \to \\ \textbf{error} \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} \rangle \, [j] : u) = \textbf{error} \qquad \text{RED-LETSUB-RAWDATA-ERROR}$$

$$((\Gamma', v \colon u' = \langle \mathrm{C} \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid v[j] \colon u \leftarrow h;\ e) \to \\ (\Gamma', v \colon u' = \langle \mathrm{C}' \rangle \mid \mathcal{C} \mid e) \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} \rangle \, [j] : u \leftarrow h) = \langle \mathrm{C}' \rangle \qquad \text{RED-SETSUB-RAWDATA}$$

$$((\Gamma', v \colon u' = \langle \mathrm{C} \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid v[j] \colon u \leftarrow h;\ e) \to \\ \textbf{error} \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} \rangle \, [j] : u \leftarrow h) = \textbf{error} \qquad \text{RED-SETSUB-RAWDATA-ERROR}$$

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

$$((\Gamma', v_2 \colon u' = \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \to \\ (\Gamma, v_1 : u = h \mid \mathcal{C} \mid e) \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \, [j] : u) = h \qquad \text{RED-LETSUB-FRAME}$$

$$((\Gamma', v_2 \colon u' = \langle \mathrm{C} : \textbf{frame}(tv[\lceil t_i \rceil_1^m]) \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \to \\ \textbf{error} \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \, [j] : u) = \textbf{error} \qquad \text{RED-LETSUB-FRAME-ERROR}$$

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

$$((\Gamma', v \colon u' = \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid v[j] \colon u \leftarrow h;\ e) \to \\ (\Gamma', v \colon u' = \langle \mathrm{C}' : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \mid \mathcal{C} \mid e) \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \, [j] : u \leftarrow h) \\ = \langle \mathrm{C}' : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \qquad \text{RED-SETSUB-FRAME}$$

$$((\Gamma', v \colon u' = \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle) \textbf{ as } \Gamma \mid \mathcal{C} \mid v[j] \colon u \leftarrow h;\ e) \to \\ \textbf{error} \\ \textbf{when}\, \textbf{runtime}(\Gamma \mid \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \, [j] : u \leftarrow h) = \textbf{error} \qquad \text{RED-SETSUB-FRAME-ERROR}$$

5.8 Global variable expressions

Global operations fetch and assign values to global labels.

$$((\Gamma', l \colon u = h) \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v \colon u = \textbf{global}\, l \textbf{ in } e) \to (\Gamma, v \colon u = h \mid \mathcal{C} \mid e) \qquad \text{RED-LETGLOBAL}$$

$$((\Gamma', l \colon u = h') \textbf{ as } \Gamma \mid \mathcal{C} \mid \textbf{global}\, l \colon u \leftarrow h;\ e) \to (\Gamma', l \colon u = h \mid \mathcal{C} \mid e) \qquad \text{RED-SETGLOBAL}$$

Chapter 6

Basic FIR properties

In this chapter, we develop a few basic lemmas that are needed to prove the preservation and progress (type-safety) 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 Γ 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 Γ 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 $\Gamma \vdash \diamond$ 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* $\Gamma, d \vdash \diamond$ *for some definition or declaration* d*, and* $dom(d) \cap FV(\Gamma) = \emptyset$*, then* $\Gamma \vdash \diamond$.

We prove this by induction on the length of the proof of $\Gamma, d \vdash \diamond$. 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 $tv\colon k$, $v\colon t$, or $l\colon t$ and the well-formedness proof ends with one of the following rules.

$$\frac{\Gamma \vdash \diamond}{\Gamma, tv\colon k \vdash \diamond} \quad \textsc{Thin-Type}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash u : \Omega}{\Gamma, v\colon u \vdash \diamond} \quad \textsc{Thin-Var}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash u : \Omega}{\Gamma, l\colon u \vdash \diamond} \quad \textsc{Thin-Global}$$

In each of these cases, $\Gamma \vdash \diamond$ follows directly from the premises of the rule. $\triangle$

Definitions Suppose d is a definition $tv\colon k = \mathit{tydef}$, $v\colon t = b$, or $l\colon t = h$, and the well-formedness proof ends with one of the following rules.

$$\frac{\Gamma, tv\colon k \vdash \mathit{tydef} : k \quad \Gamma, tv\colon k \vdash \diamond}{\Gamma, tv\colon k = \mathit{tydef} \vdash \diamond} \quad \textsc{Thin-Type-Def}$$

$$\frac{\Gamma, v\colon u \vdash b : u \quad \Gamma, v\colon u \vdash \diamond}{\Gamma, v\colon u = b \vdash \diamond} \quad \textsc{Thin-Var-Def}$$

$$\frac{\Gamma, l\colon u \vdash h : u \quad \Gamma, l\colon u \vdash \diamond}{\Gamma, l\colon u = h \vdash \diamond} \quad \textsc{Thin-Global-Def}$$

The judgment $\Gamma \vdash \diamond$ follows by induction and the premises $\Gamma, tv\colon k \vdash \diamond$, $\Gamma, v\colon t \vdash \diamond$, and $\Gamma, l\colon t \vdash \diamond$. $\triangle$

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

Declarations If d' is a declaration $tv\colon k$, $v\colon t$, or $l\colon t$, then the well-formedness proof ends with one of the following rules.

$$\frac{\Gamma', d \vdash \diamond}{\Gamma', d, tv\colon k \vdash \diamond} \quad \textsc{Thin-Type}$$

$$\frac{\Gamma', d \vdash \diamond \quad \Gamma', d \vdash u : \Omega}{\Gamma', d, v\colon u \vdash \diamond} \quad \textsc{Thin-Var}$$

$$\frac{\Gamma', d \vdash \diamond \quad \Gamma', d \vdash u : \Omega}{\Gamma', d, l\colon u \vdash \diamond} \quad \textsc{Thin-Global}$$

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

Definitions If d' is a definition $tv\colon k = t$, $v\colon t = b$, or $l\colon t = h$, then the well-formedness proof ends with one of the following rules.

$$\frac{\Gamma', d, tv\colon k \vdash \mathit{tydef} : k \quad \Gamma', d, tv\colon k \vdash \diamond}{\Gamma', d, tv\colon k = \mathit{tydef} \vdash \diamond} \quad \textsc{Thin-Type-Def}$$

$$\frac{\Gamma', d, v\colon u \vdash b : u \quad \Gamma', d, v\colon u \vdash \diamond}{\Gamma', d, v\colon u = b \vdash \diamond} \quad \textsc{Thin-Var-Def}$$

$$\frac{\Gamma', d, l\colon u \vdash h : u \quad \Gamma', d, l\colon u \vdash \diamond}{\Gamma', d, l\colon u = h \vdash \diamond} \quad \textsc{Thin-Global-Def}$$

Let d'' be the declaration $tv\colon k$, $v\colon t$, or $l\colon t$ that corresponds to the definition d'. We can infer that $\Gamma', d'' \vdash \diamond$ 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 $\Gamma', d \vdash \diamond$. △

This concludes the proof of the WELL-FORMED SUB-CONTEXTS Lemma 6.1.1. □

Lemma 6.1.2 WELL-FORMED CONTEXT TYPES *If $\Gamma \vdash \diamond$ then for each declaration $v\colon t$, $l\colon t$, or each definition $v\colon t = b$, $l\colon t = b$ in Γ, the term t is a well-formed type.*

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

Empty context If Γ is empty, it contains no declarations or definitions, and the result follows trivially. △

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

$$\frac{\Gamma \vdash \diamond}{\Gamma, tv\colon k \vdash \diamond} \quad \textsc{Thin-Type}$$

$$\frac{\Gamma, tv\colon k \vdash \mathit{tydef} : k \quad \Gamma, tv\colon k \vdash \diamond}{\Gamma, tv\colon k = \mathit{tydef} \vdash \diamond} \quad \textsc{Thin-Type-Def}$$

By induction, we can infer that each variable and global definition in Γ is defined over a well-formed type. Since the declaration $tv\colon k$ and definition $tv\colon k = t$ 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.

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash u : \Omega}{\Gamma, v\colon u \vdash \diamond} \quad \textsc{Thin-Var}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash u : \Omega}{\Gamma, l\colon u \vdash \diamond} \quad \textsc{Thin-Global}$$

Using the induction hypothesis on the premise $\Gamma \vdash \diamond$ we can conclude that each variable and global definition in Γ 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.

$$\frac{\Gamma, v\colon u \vdash b : u \quad \Gamma, v\colon u \vdash \diamond}{\Gamma, v\colon u = b \vdash \diamond} \quad \textsc{Thin-Var-Def}$$

$$\frac{\Gamma, l\colon u \vdash h : u \quad \Gamma, l\colon u \vdash \diamond}{\Gamma, l\colon u = h \vdash \diamond} \quad \textsc{Thin-Global-Def}$$

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

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

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

Lemma 6.1.3 WELL-FORMED CONTEXTS *A proof of the judgment* $\Gamma \vdash \diamond$ *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 $J \neq \diamond$, 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.

$$\frac{\Gamma, tv\colon k \vdash \mathit{tydef} : k \quad \Gamma, tv\colon k \vdash J}{\Gamma, tv\colon k = \mathit{tydef} \vdash J} \quad \text{THIN-TYPE-DEF}$$

$$\frac{\Gamma, v\colon u \vdash b : u \quad \Gamma, v\colon u \vdash J}{\Gamma, v\colon u = b \vdash J} \quad \text{THIN-VAR-DEF}$$

$$\frac{\Gamma, l\colon u \vdash h : u \quad \Gamma, l\colon u \vdash J}{\Gamma, l\colon u = h \vdash J} \quad \text{THIN-GLOBAL-DEF}$$

$$\frac{\Gamma, \lceil \alpha_i\colon \omega \rceil_1^m \vdash \mathit{tydef}_s : k_s}{\Gamma \vdash \Lambda\alpha_1, \ldots, \alpha_m.\mathit{tydef}_s : \omega^m \to k_s} \quad \text{WF-TYDEF}$$

$$\frac{\Gamma, \lceil v_i\colon u_i \rceil_1^n \vdash e : t}{\Gamma \vdash \lambda v_1, \ldots, v_n.e : (u_1, \ldots, u_n) \to t} \quad \text{TY-STOREFUN-MONO}$$

$$\frac{\Gamma, \lceil \alpha_i\colon \omega \rceil_1^m \vdash \lambda v_1, \ldots, v_n.e : (u_1, \ldots, u_n) \to t}{\Gamma \vdash \Lambda\alpha_1, \ldots, \alpha_m.\lambda v_1, \ldots, v_n.e : \forall\alpha_1, \ldots, \alpha_m.(u_1, \ldots, u_n) \to t} \quad \text{TY-STOREFUN-POLY}$$

$$\frac{\Gamma, \lceil \alpha_i\colon \omega \rceil_1^m \vdash t : \Omega}{\Gamma \vdash \exists\alpha_1, \ldots, \alpha_m.t : \omega} \quad \text{WF-ST-TYEXISTS}$$

$$\frac{\Gamma, \lceil \alpha_i\colon \omega \rceil_1^m \vdash ((t_1, \ldots, t_n) \to u) : \omega}{\Gamma \vdash \forall\alpha_1, \ldots, \alpha_m.((t_1, \ldots, t_n) \to u) : \omega} \quad \text{WF-ST-TYFUN-POLY}$$

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \{\} \neq s_{union} = \bigcup_{i=1}^n s_i \quad \lceil \Gamma, v\colon \mathbf{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t \rceil_1^n}{\Gamma, v\colon \mathbf{union}(tv[t_1, \ldots, t_m], s_{union}) \vdash \mathbf{match}\, v\ \mathbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t} \quad \text{TY-MATCH-UNION}$$

Subcontexts If the proof ends with the rule WF-TYDEF, TY-STOREFUN-POLY, WF-ST-TYEXISTS, or WF-ST-TYFUN-POLY, we can conclude by induction that $\Gamma, \lceil \alpha_i\colon \omega \rceil_1^m \vdash \diamond$.

Similarly, if the proof ends with the rule TY-STOREFUN-MONO, we can conclude by induction that $\Gamma, \lceil v_i\colon t_i \rceil_1^m \vdash \diamond$.

The result $\Gamma \vdash \diamond$ follows directly by application of the WELL-FORMED SUB-CONTEXTS Lemma 6.1.1. $\triangle$

Definitions From the three cases THIN-TYPE-DEF, THIN-VAR-DEF, and THIN-GLOBAL-DEF, let the definitions and declarations be defined as in the following table.

decl	*def*	Rule
$tv\colon k$	$tv\colon k = t$	THIN-TYPE-DEF
$v\colon t$	$v\colon t = b$	THIN-VAR-DEF
$l\colon t$	$l\colon t = h$	THIN-GLOBAL-DEF

We can conclude $\Gamma, \mathit{decl} \vdash \diamond$ 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 $\Gamma, \mathit{def} \vdash \diamond$. $\triangle$

Unions For TY-MATCH-UNION, we can conclude from the premise $\lceil \Gamma, v \colon \mathbf{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t \rceil_1^n$, that the context $\Gamma, v \colon \mathbf{union}(tv[t_1, \ldots, t_m], s_i)$ is well-formed for all of the match cases $\lceil s_i \mapsto e_i \rceil_1^n$. 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 $\Gamma, v \colon \mathbf{union}(tv[t_1, \ldots, t_m], s_1) \vdash \diamond$.

From the WELL-FORMED CONTEXT TYPES Lemma 6.1.2, the type $\mathbf{union}(tv[t_1, \ldots, t_m], s_1)$ must be well-formed. That is, there is a proof $\Gamma \vdash \mathbf{union}(tv[t_1, \ldots, t_m], s_1) : k_s$. The only rule that can be used directly to prove well-formedness is the WF-ST-TYUNION rule.

$$\frac{\mathit{set}_I \subseteq \{0 \ldots n-1\} \quad \Gamma \vdash \lceil t_i : \omega \rceil_1^m \quad \Gamma \vdash tv[t_1^2, \ldots, t_m^2] : \omega_{\mathit{union}[n]}}{\Gamma \vdash \mathbf{union}(tv[t_1, \ldots, t_m], \mathit{set}_I) : \omega}$$

The only condition on set_I is $\mathit{set}_I \subseteq \{0 \ldots n-1\}$, and we can infer the type $\mathbf{union}(tv[t_1, \ldots, t_m], s_{\mathit{union}})$ is also well-formed. We can infer from the WELL-FORMED SUB-CONTEXTS Lemma 6.1.1 that $\Gamma \vdash \diamond$, and we can conclude that $\Gamma, v \colon \mathbf{union}(tv[t_1, \ldots, t_m], s_{\mathit{union}}) \vdash \diamond$ from the THIN-VAR rule.

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash u : \Omega}{\Gamma, v \colon u \vdash \diamond}$$

$\triangle$

This concludes the proof of the WELL-FORMED CONTEXTS Lemma 6.1.3. $\square$

Lemma 6.1.4 WELL-FORMED CONTEXT VALUES *If $\Gamma \vdash \diamond$, then every value in each definition in Γ is well-typed. That is, all of the following hold.*

- *For each type definition $tv \colon k = t$ in Γ, $\Gamma \vdash t : k$.*
- *For each variable definition $v \colon t = b$ in Γ, $\Gamma \vdash b : t$.*
- *For each global definition $l \colon t = h$ in Γ, $\Gamma \vdash h : t$.*

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

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.

$$\frac{\Gamma \vdash \diamond}{\Gamma, tv \colon k \vdash \diamond} \quad \text{THIN-TYPE}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash u : \Omega}{\Gamma, v \colon u \vdash \diamond} \quad \text{THIN-VAR}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash u : \Omega}{\Gamma, l \colon u \vdash \diamond} \quad \text{THIN-GLOBAL}$$

We can conclude that each definition in Γ is well-formed from the premise $\Gamma \vdash \diamond$. $\triangle$

Definitions Suppose the well-formedness proof ends with one of the definition forms.

$$\frac{\Gamma, tv\colon k \vdash \mathit{tydef} : k \quad \Gamma, tv\colon k \vdash \diamond}{\Gamma, tv\colon k = \mathit{tydef} \vdash \diamond} \quad \text{THIN-TYPE-DEF}$$

$$\frac{\Gamma, v\colon u \vdash b : u \quad \Gamma, v\colon u \vdash \diamond}{\Gamma, v\colon u = b \vdash \diamond} \quad \text{THIN-VAR-DEF}$$

$$\frac{\Gamma, l\colon u \vdash h : u \quad \Gamma, l\colon u \vdash \diamond}{\Gamma, l\colon u = h \vdash \diamond} \quad \text{THIN-GLOBAL-DEF}$$

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 φ 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 $\Gamma \vdash \mathbf{let}\, v\colon u = a \;\mathbf{in}\; e : t$, we would argue that the final rule in the proof *must be* the TY-LETATOM rule, because *no other rule* applies.

$$\frac{\Gamma \vdash a : u \quad \Gamma, v\colon u \vdash e : t}{\Gamma \vdash \mathbf{let}\, v\colon u = a \;\mathbf{in}\; e : t}$$

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

$$\frac{\Gamma \vdash \varphi : t_1}{\Gamma, tv\colon k \vdash \varphi : t_1} \quad \text{THIN-TYPE}$$

$$\frac{\Gamma \vdash \varphi : t_1 \quad \Gamma \vdash t_2 : \Omega}{\Gamma, v\colon t_2 \vdash \varphi : t_1} \quad \text{THIN-VAR}$$

$$\frac{\Gamma \vdash \varphi : t_1 \quad \Gamma \vdash t_2 : \Omega}{\Gamma, l\colon t_2 \vdash \varphi : t_1} \quad \text{THIN-GLOBAL}$$

$$\frac{\Gamma, tv\colon k \vdash \mathit{tydef} : k \quad \Gamma, tv\colon k \vdash \varphi : t_1}{\Gamma, tv\colon k = \mathit{tydef} \vdash \varphi : t_1} \quad \text{THIN-TYPE-DEF}$$

$$\frac{\Gamma, v\colon t_2 \vdash b : t_2 \quad \Gamma, v\colon t_2 \vdash \varphi : t_1}{\Gamma, v\colon t_2 = b \vdash \varphi : t_1} \quad \text{THIN-VAR-DEF}$$

$$\frac{\Gamma, l\colon t_2 \vdash h : t_2 \quad \Gamma, l\colon t_2 \vdash \varphi : t_1}{\Gamma, l\colon t_2 = h \vdash \varphi : t_1} \quad \text{THIN-GLOBAL-DEF}$$

$$\frac{\Gamma \vdash t_1 = t_2 : k \quad \Gamma \vdash \varphi : t_2}{\Gamma \vdash \varphi : t_1} \quad \text{TY-SUBST}$$

Figure 6.1: Thinning and subsitution 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 φ 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 $r \notin R$ with conclusion $\widehat{\Gamma} \vdash \varphi : t'$ for some context $\widehat{\Gamma}$ where $\widehat{\Gamma} \vdash t = t' : \Omega$. Furthermore, for each premise of r of the form $\widehat{\Gamma}, \Delta \vdash J$, 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 $r \notin R$. The result follows trivially. △

Declarations Suppose the final rule in is the application of one of the rules THIN-TYPE, THIN-VAR, or THIN-GLOBAL. These rules are similar; we illustrate the argument with the THIN-VAR rule.

$$\frac{\Gamma \vdash \varphi : t \quad \Gamma \vdash u : \Omega}{\Gamma, v\colon u \vdash \varphi : t}$$

We can conclude from the induction hypothesis and the premise $\Gamma \vdash \varphi : t$ that the proof contains the use of a rule $r \notin R$ for the abbreviated context $\widehat{\Gamma} = \Gamma$. For any derivation $\widehat{\Gamma}, \Delta \vdash J$, the context is well-formed $\widehat{\Gamma}, \Delta \vdash \diamond$ by the WELL-FORMED CONTEXTS Lemma 6.1.3, and we can infer that $\widehat{\Gamma}, \Delta \vdash u : \Omega$ from the premise $\widehat{\Gamma}, u : \Omega \vdash$ and application of the thinning rules. The derivation $\Gamma, v\colon u, \Delta \vdash J$ follows by re-applying the THIN-VAR rule. △

Definitions Suppose the final rule in the proof is the application of one of the rules THIN-TYPE-DEF, THIN-VAR-DEF, or THIN-GLOBAL-DEF. These rules are similar; we illustrate the argument with the THIN-VAR-DEF rule.

$$\frac{\Gamma, v\colon u \vdash b : u \quad \Gamma, v\colon u \vdash \varphi : t}{\Gamma, v\colon u = b \vdash \varphi : t}$$

We can conclude from the induction hypothesis and the premise $\Gamma, v\colon t \vdash \varphi : t$ that the proof contains the use of a rule $r \notin R$ for the abbreviated context $\widehat{\Gamma} = \Gamma, v\colon t$. For any derivation $\widehat{\Gamma}, \Delta \vdash J$, the context is well-formed $\widehat{\Gamma}, \Delta \vdash \diamond$ by the WELL-FORMED CONTEXTS Lemma 6.1.3, and we can infer that $\widehat{\Gamma}, \Delta \vdash b : t$, from the premise $\widehat{\Gamma} \vdash b : t$ and repeated application of the thinning rules. Thus, for any derivation $\widehat{\Gamma}, \Delta \vdash J$, we can re-apply the THIN-VAR-DEF rule to conclude that $\Gamma, v\colon t = b \vdash J$. △

Substitution Suppose the final rule in the proof is an application of TY-SUBST.

$$\frac{\Gamma \vdash t = u : k \quad \Gamma \vdash \varphi : u}{\Gamma \vdash \varphi : t}$$

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 $r \notin R$, and the context Γ is unchanged. △

This concludes the PROOF INDUCTION Lemma 6.2.1. □

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 $\mathit{typeof}(\Gamma, a)$ that produces the type of the atom a, given the context Γ and the atom.

a	$\mathit{typeof}(\Gamma, a)$	Rule
$\mathbf{enum}_i(j)$	$\mathbf{enum}_i$	TY-ATOMENUM
$\mathbf{int}(i)$	$\mathbb{Z}_{31}$	TY-ATOMINT
$\mathbf{rawint}^{sign}_{pre}(r)$	$\mathbb{Z}^{sign}_{pre}$	TY-ATOMRAWINT
$\mathbf{float}_{fpre}(x)$	$\mathbb{R}_{fpre}$	TY-ATOMFLOAT
$\mathbf{label}(tv, l^f, l^{sf}, r_{off})$	$\mathbb{Z}^{\mathbf{signed}}_{32}$	TY-ATOMLABEL
$\mathbf{sizeof}(tv_1, \ldots, tv_n, r_{size})$	$\mathbb{Z}^{\mathbf{signed}}_{32}$	TY-ATOMSIZEOF
v	$\Gamma(v)$	TY-ATOMVAR
$\mathbf{const}[t](tv, i)$	t	TY-ATOMCONST
$\mathbf{ty\_apply}[t](a, t_1, \ldots, t_n)$	t	TY-ATOMTYAPPLY
$\mathbf{ty\_pack}[t](v, t_1, \ldots, t_n)$	t	TY-ATOMTYPACK
$\mathbf{ty\_unpack}(v)$	$(\Gamma(v))[\lceil v.i/\alpha_i \rceil_0^{m-1}]$	TY-ATOMTYUNPACK
$\mathit{unop}\ a$	$\mathbf{res}(\mathit{unop})$	TY-ATOMUNOP
$a_1\ \mathit{binop}\ a_2$	$\mathbf{res}(\mathit{binop})$	TY-ATOMBINOP

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

Lemma 6.3.2 UNIQUENESS OF EXPRESSION TYPES *If* $\Gamma \vdash e_r : t$ *then* $t = \mathbf{enum}_0$.

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 TY-LETATOM, TY-LETEXT, TY-LETSUB-TUPLE, TY-LETSUB-ARRAY, TY-LETSUB-RAWDATA, TY-LETSUB-UNION, TY-LETSUB-FRAME, TY-ALLOC, and TY-LETGLOBAL. These all have similar proofs. We illustrate with the proof for TY-LETATOM.

Suppose $e_r = \mathbf{let}\, v\colon u = a\ \mathbf{in}\ e$. The proof of typing for e_r must use the rule TY-LETATOM.

$$\frac{\widehat{\Gamma} \vdash a : u \quad \widehat{\Gamma}, v\colon u \vdash e : t}{\widehat{\Gamma} \vdash \mathbf{let}\, v\colon u = a\ \mathbf{in}\ e : t}$$

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

TailCall Suppose $e_r = a_f(a_1, \ldots, a_n)$. The proof of typing for e_r must use the rule TY-TAILCALL.

$$\frac{\widehat{\Gamma} \vdash a_f : (u_1, \ldots, u_n) \to \mathbf{enum}_0 \quad \widehat{\Gamma} \vdash \lceil a_i : u_i \rceil_1^n}{\widehat{\Gamma} \vdash a_f(a_1, \ldots, a_n) : \mathbf{enum}_0}$$

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

Special-calls Suppose $e_r = \mathbf{special}\, \mathit{tailop}$. The proof of typing for e_r must use the TY-SPECIAL-CALL rule.

$$\frac{\widehat{\Gamma} \vdash \mathit{tailop} : \mathbf{special}\, t}{\widehat{\Gamma} \vdash \mathbf{special}\, \mathit{tailop} : t}$$

There are four rules that can prove the premise $\widehat{\Gamma} \vdash \mathit{tailop} : \mathbf{special}\, t$: TY-SYSMIGRATE, TY-ATOMIC, TY-ATOMICCOMMIT, and TY-ATOMICROLLBACK. In each of these cases $t = \mathbf{enum}_0$. $\triangle$

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 $\Gamma \vdash e_r : t$ has the following form.

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \{\} \neq s_{union} = \bigcup_{i=1}^n s_i \quad \left\lceil \widehat{\Gamma}, v \colon \mathbf{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t \right\rceil_1^n}{\widehat{\Gamma}, v \colon \mathbf{union}(tv[t_1, \ldots, t_m], s_{union}) \vdash \mathbf{match}\, v\, \mathbf{with}\, \lceil s_i \mapsto e_i \rceil_1^n : t}$$

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

$$\frac{\widehat{\Gamma} \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \widehat{\Gamma} \vdash a_3 : u \quad \widehat{\Gamma} \vdash e : t}{\widehat{\Gamma} \vdash a_1[\mathbf{offset}_c^{\mathbf{const}}(j)] \colon u \leftarrow a_3;\ e : t}$$

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

Assignment The type judgment for an assignment operation TY-SETGLOBAL has the following form.

$$\frac{\widehat{\Gamma} \vdash l : u \quad \widehat{\Gamma} \vdash a : u \quad \widehat{\Gamma} \vdash e : t}{\widehat{\Gamma} \vdash \mathbf{global}\, l \colon u \leftarrow a;\ e : t}$$

The premise $\widehat{\Gamma} \vdash e : t$ requires that $t = \mathbf{enum}_0$ 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 $A[a] = \mathit{unop}\, A'[a]$. The type judgment must use the Ty-AtomUnop rule.

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

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

$$\Gamma_r \vdash a : t_a.$$

$\triangle$

Binary operators Suppose $A[a] = A'[a]\ \mathit{binop}\ a'$. the type judgment must use the Ty-AtomBinop rule.

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

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

$$\Gamma_r \vdash a : t_a.$$

The case $A[a] = h\ \mathit{binop}\ A'[a]$ is similar. $\triangle$

This concludes the Atom typing Lemma 6.4.1. $\square$

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.

$$\begin{array}{rcl} S & ::= & \textbf{migrate}\ [i, A, a_o]\ a_{fun}(a_1, \ldots, a_n) \\ & | & \textbf{migrate}\ [i, h_p, A]\ a_{fun}(a_1, \ldots, a_n) \\ & | & \textbf{migrate}\ [i, h_p, h_o]\ A(a_1, \ldots, a_n) \\ & | & \textbf{migrate}\ [i, h_p, h_o]\ h_{fun}(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \end{array}$$

The type judgment for S must use the rule Ty-SysMigrate.

$$\frac{\begin{array}{cc} & \widehat{\Gamma_r} \vdash a_{ptr} : \textbf{data} \\ j \in \mathbb{Z}_{31} & \widehat{\Gamma_r} \vdash a_{off} : \mathbb{Z}_{32}^{\textbf{signed}} \\ \widehat{\Gamma_r} \vdash \lceil a_i : t_i \rceil_1^n & \widehat{\Gamma_r} \vdash a_f : (t_1, \ldots, t_n) \rightarrow \textbf{enum}_0 \end{array}}{\widehat{\Gamma_r} \vdash \textbf{migrate}\ [j, a_{ptr}, a_{off}]\ a_f(a_1, \ldots, a_n) : \textbf{special}\,\textbf{enum}_0}$$

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

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

From the Atom typing Lemma 6.4.1, we can infer that there is some type t_a for which $\Gamma_r \vdash a : t_a$. $\triangle$

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

$$\begin{array}{rcl} S & ::= & \textbf{atomic}\ A(a_{const}, a_1, \ldots, a_n) \\ & | & \textbf{atomic}\ v(A, a_1, \ldots, a_n) \\ & | & \textbf{atomic}\ v(h_{const}, h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \end{array}$$

The type judgment for S must use the rule Ty-Atomic.

$$\frac{\widehat{\Gamma_r} \vdash a_{const} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \widehat{\Gamma_r} \vdash \lceil a_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash a_f : (\mathbb{Z}_{32}^{\textbf{signed}}, t_1, \ldots, t_n) \rightarrow \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{atomic}\ a_f(a_{const}, a_1, \ldots, a_n) : \textbf{special}\,\textbf{enum}_0}$$

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

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

From the Atom typing Lemma 6.4.1, we can infer that there is some type t_a for which $\Gamma_r \vdash a : t_a$. $\triangle$

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

$$\begin{array}{rcl} S & ::= & \textbf{rollback}\ [A, a_{const}] \\ & | & \textbf{rollback}\ [h_{level}, A] \end{array}$$

The type judgment for S must use the rule Ty-AtomicRollback.

$$\frac{\widehat{\Gamma_r} \vdash a_{level} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \widehat{\Gamma_r} \vdash a_{const} : \mathbb{Z}_{32}^{\textbf{signed}}}{\widehat{\Gamma_r} \vdash \textbf{rollback}\ [a_{level}, a_{const}] : \textbf{special enum}_0}$$

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

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

$$\begin{array}{rcl} S & ::= & \textbf{commit}\ [A]\ a_{fun}(a_1, \ldots, a_n) \\ & | & \textbf{commit}\ [h_{level}]\ A(a_1, \ldots, a_n) \\ & | & \textbf{commit}\ [h_{level}]\ v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \end{array}$$

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

$$\frac{\widehat{\Gamma_r} \vdash a_{level} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \widehat{\Gamma_r} \vdash \lceil a_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash a_f : (t_1, \ldots, t_n) \to \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{commit}\ [a_{level}]\ a_f(a_1, \ldots, a_n) : \textbf{special enum}_0}$$

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

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

From the Atom typing Lemma 6.4.1, we can infer that there is some type t_a for which $\Gamma_r \vdash a : t_a$. △

This concludes the Special-call typing Lemma 6.4.2. □

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

$$\frac{\widehat{\Gamma_r} \vdash \lceil a_i : u_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash \langle u_1, \ldots, u_n \rangle_c : \omega}{\widehat{\Gamma_r} \vdash \langle a_1, \ldots, a_n \rangle_c : \textbf{alloc}\ \langle u_1, \ldots, u_n \rangle_c}$$

From the premise $\Gamma_r \vdash a_i : t_i$, we can infer that $\Gamma_r \vdash A[a] : t_i$. 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 union Suppose $L = \textbf{union}(tv[h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n], i) : t$. The type judgment must use the rule Ty-Alloc-Union.

$$\frac{\widehat{\Gamma_r} \vdash tv_j(a_1, \ldots, a_k) : \textbf{union}(tv[t_1, \ldots, t_m], \{j\})}{\widehat{\Gamma_r} \vdash \textbf{union}(tv[a_1, \ldots, a_k], j) : \textbf{alloc union}(tv[t_1, \ldots, t_m], \{j\})}$$

The derivation of the premise must use the store-typing rule TY-STOREUNION for unions.

$$\frac{\begin{array}{l}\widehat{\Gamma_r}, \lceil \alpha_i \colon \omega = t'_i \rceil_1^m \vdash \lceil a_i : u_i \rceil_1^k \\ \widehat{\Gamma_r} \vdash \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) : \Omega \\ \widehat{\Gamma_r} \vdash tv[t'_1, \ldots, t'_m] = \vec{t}_0 + \cdots + \vec{t}_{j-1} + \langle u_1, \ldots, u_k \rangle + \vec{t}_{j+1} + \cdots + \vec{t}_{n-1} : \omega_{union[n]}\end{array}}{\widehat{\Gamma_r} \vdash tv_j(a_1, \ldots, a_k) : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\})}$$

From the premise $\Gamma_r \vdash a_i : t_i$, we can infer that $\Gamma_r \vdash A[a] : t_i$. 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.

$$\begin{array}{rcl} L & ::= & \mathbf{array}(A, a_{init}) : t \\ & | & \mathbf{array}(h_{size}, A) : t \end{array}$$

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

$$\frac{\widehat{\Gamma_r} \vdash a_{size} : \mathbf{offset} \quad \widehat{\Gamma_r} \vdash a_{init} : u}{\widehat{\Gamma_r} \vdash \mathbf{array}(a_{size}, a_{init}) : \mathbf{alloc}\, u\ \mathbf{array}}$$

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

- If $A = a_{size}$, then $t_A = \mathbf{offset}$.
- If $A = a_{init}$, 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 $L = \mathbf{malloc}(A) : t$. The type judgment must use the rule TY-ALLOC-MALLOC.

$$\frac{\widehat{\Gamma_r} \vdash A[a] : \mathbf{offset}}{\widehat{\Gamma_r} \vdash \mathbf{malloc}(A[a]) : \mathbf{alloc\, data}}$$

From the premise $\Gamma_r \vdash A[a] : \mathbf{offset}$, 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 $E = \mathbf{let}\, v \colon t = A\ \mathbf{in}\ e$. The type judgment must use the TY-LETATOM rule.

$$\frac{\widehat{\Gamma_r} \vdash A[a] : u \quad \widehat{\Gamma_r}, v \colon u \vdash e : t_r}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v \colon u = A[a]\ \mathbf{in}\ e : t_r}$$

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 $\Gamma_r \vdash a : t_a$. △

LetExt Suppose $E = \textbf{let}\, v\colon t = (\text{“}s\text{”} : t_s)(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_{n-1})\ \textbf{in}\ e$. the type judgment must use the TY-LETEXT rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil a_i : u_i \rceil_1^n \quad \widehat{\Gamma_r}, v\colon u \vdash e : t_r \quad \widehat{\Gamma_r} \vdash [\![\text{“s”}]\!] : (u_1, \ldots, u_n) \to u}{\widehat{\Gamma_r} \vdash \textbf{let}\, v\colon u = (\text{“}s\text{”} : (u_1, \ldots, u_n) \to u)(a_1, \ldots, a_n)\ \textbf{in}\ e : t_r}$$

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 $\Gamma_r \vdash a : t_a$. △

TailCall Suppose E is one of the following.

$$\begin{array}{rcl} E & ::= & A(a_1, \ldots, a_n) \\ & | & v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_{n-1}) \end{array}$$

The type judgment must use the TY-TAILCALL rule.

$$\frac{\widehat{\Gamma_r} \vdash a_f : (u_1, \ldots, u_n) \to \textbf{enum}_0 \quad \widehat{\Gamma_r} \vdash \lceil a_i : u_i \rceil_1^n}{\widehat{\Gamma_r} \vdash a_f(a_1, \ldots, a_n) : \textbf{enum}_0}$$

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

- If $A = a_f$, then $t_A = (u_1, \ldots, u_n) \to \textbf{enum}_0$.
- 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 $\Gamma_r \vdash a : t_a$. △

Special Call Suppose $E = \textbf{special}\, S$. The type judgment must use the TY-SPECIAL-CALL rule.

$$\frac{\widehat{\Gamma_r} \vdash S : \textbf{special}\, t_r}{\widehat{\Gamma_r} \vdash \textbf{special}\, S : t_r}$$

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

Match Suppose $E = \textbf{match}\, A\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n$. The type judgment must use the one of the match rules.

- If the rule is TY-MATCH-INT

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{31} \quad \widehat{\Gamma_r} \vdash A : \mathbb{Z}_{31} \quad \widehat{\Gamma_r} \vdash \lceil e_i : t_r \rceil_1^n}{\widehat{\Gamma_r} \vdash \textbf{match}\, A\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t_r}$$

  then we can infer from the premise $A : \mathbb{Z}_{31}$ 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

$$\frac{j > 0 \quad \lceil s_i \in set_I \rceil_1^n \quad \bigcup_{i=1}^n s_i = \textbf{enum}_j \quad \widehat{\Gamma_r} \vdash A : \textbf{enum}_j \quad \widehat{\Gamma_r} \vdash \lceil e_i : t_r \rceil_1^n}{\widehat{\Gamma_r} \vdash \textbf{match}\, A\ \textbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t_r}$$

  then we can infer from the premise $A : \textbf{enum}_i$ 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-RAWINT

$$\frac{\lceil s_i \in set_R \rceil_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{pre}^{sign} \quad \widehat{\Gamma_r} \vdash A : \mathbb{Z}_{pre}^{sign} \quad \widehat{\Gamma_r} \vdash \lceil e_i : t_r \rceil_1^n}{\widehat{\Gamma_r} \vdash \textbf{match}\, A \textbf{ with } \lceil s_i \mapsto e_i \rceil_1^n : t_r}$$

  then we can infer from the premise $A : \mathbb{Z}_{pre}^{sign}$ 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

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \{\} \neq s_{union} = \bigcup_{i=1}^n s_i \quad \left\lceil \widehat{\Gamma_r}, v\colon \textbf{union}(tv[t_1, \ldots, t_m], s_i) \vdash e_i : t_r \right\rceil_1^n}{\widehat{\Gamma_r}, v\colon \textbf{union}(tv[t_1, \ldots, t_m], s_{union}) \vdash \textbf{match}\, v \textbf{ with } \lceil s_i \mapsto e_i \rceil_1^n : t_r}$$

  requires that $A = v$. We can infer from the declaration $v\colon \textbf{union}(tv[u_1, \ldots, u_m], s_{union})$ that $A : \textbf{union}(tv[u_1, \ldots, u_m], s_{union})$. 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 $E = \textbf{let}\, v = L \textbf{ in } e$. The type judgment must use the TY-ALLOC rule.

$$\frac{\widehat{\Gamma_r} \vdash L : \textbf{alloc}\, u \quad \widehat{\Gamma_r}, v\colon u \vdash e : t_r}{\widehat{\Gamma_r} \vdash \textbf{let}\, v = L \textbf{ in } e : t_r}$$

From the premise $L : \textbf{alloc}\, u$ 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.

$$\begin{array}{lcl} E & ::= & \textbf{let}\, v\colon t = A[a_2] \textbf{ in } e \\ & | & \textbf{let}\, v\colon t = h[A] \textbf{ in } e \end{array}$$

The type judgment must use one of the subscript rules.

- Suppose the rule is TY-LETSUB-RAWDATA.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : \textbf{data} \quad \widehat{\Gamma_r} \vdash a_2 : \textbf{offset} \quad \widehat{\Gamma_r}, v\colon u \vdash e : t_r}{\widehat{\Gamma_r} \vdash \textbf{let}\, v\colon u = a_1[a_2] \textbf{ in } e : t_r}$$

  We can infer from the premises $a_1 : \textbf{data}$, $a_2 : \textbf{offset}$, 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.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : u \textbf{ array} \quad \widehat{\Gamma_r} \vdash a_2 : \textbf{offset} \quad \widehat{\Gamma_r}, v\colon u \vdash e : t_r}{\widehat{\Gamma_r} \vdash \textbf{let}\, v\colon u = a_1[a_2] \textbf{ in } e : t_r}$$

  We can infer from the premises $a_1 : t_1 \textbf{ array}$, $a_2 : \textbf{offset}$, 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.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1}\rangle_c \quad \widehat{\Gamma_r}, v_1 : u \vdash e : t_r}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v_1 : u = a_1[\mathbf{offset}_c^{\mathbf{const}}(j)]\ \mathbf{in}\ e : t_r}$$

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

  - If $A = a$, then $t_A = \langle u_0, \ldots, u_{i-1}, t_1, u_{i+1}, \ldots, u_{n-1}\rangle_c$.
  - If $A = \mathbf{offset}_c^{\mathbf{const}}(i)$, 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.

$$\frac{\begin{array}{l}\widehat{\Gamma_r} \vdash a_1 : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) \\ \widehat{\Gamma_r} \vdash tv[t'_1, \ldots, t'_m] = \left(\lceil \vec{u}_i \rceil_0^{j-1} + \left\langle \lceil u'_i \rceil_0^{k-1}, u, \lceil u'_i \rceil_{k+1}^{l-1} \right\rangle + \lceil \vec{u}_i \rceil_{j+1}^{n-1}\right) : \omega_{union[n]} \\ \widehat{\Gamma_r}, v \colon u \vdash e : t_r\end{array}}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v \colon u = a_1[\mathbf{offset}_{\mathbf{normal}}^{\mathbf{const}}(k)]\ \mathbf{in}\ e : t_r}$$

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

  - If $A = a_1$, then $t_A = \mathbf{union}(tv[t'_1, \ldots, t'_k], \{j\})$.
  - If $A = \mathbf{offset}_{\mathbf{normal}}^{\mathbf{const}}(i)$, 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.

$$\frac{\begin{array}{l} r_{\mathit{off}} \in \mathbb{Z}_{32}^{\mathbf{signed}} \\ \widehat{\Gamma_r} \vdash a_1 : \mathbf{frame}(tv[t'_1, \ldots, t'_m]) \\ \widehat{\Gamma_r} \vdash tv[t'_1, \ldots, t'_m] = \left\{ l^f : \{ l^r : u; \cdots \} ; \cdots \right\} : \omega_{frame} \\ \widehat{\Gamma_r}, v \colon u \vdash e : t_r \end{array}}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v \colon u = a_1[\mathbf{label}(tv, l^f, l^r, r_{\mathit{off}})]\ \mathbf{in}\ e : t_r}$$

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

  - If $A = a_1$, then $t_A = \mathbf{frame}(tv[t'_1, \ldots, t'_m])$. 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 $A = \mathbf{label}(tv, l^f, l^{sf}, i_{\mathit{off}})$, is not allowed, because it is not a valid context.

$\triangle$

Assignment Suppose E is one of the following forms.

$$\begin{array}{rcl} E & ::= & A[a_2] \colon t \leftarrow a_3;\ e \\ & | & v[A] \colon t \leftarrow a_3;\ e \\ & | & v[h] \colon t \leftarrow A;\ e \end{array}$$

The type judgment must use one of the subscript assignment rules.

- Suppose the rule is Ty-SetSub-RawData.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : \mathbf{data} \quad \widehat{\Gamma_r} \vdash a_2 : \mathbf{offset} \quad \widehat{\Gamma_r} \vdash a_3 : u \quad \widehat{\Gamma_r} \vdash e : t_r}{\widehat{\Gamma_r} \vdash a_1[a_2] \colon u \leftarrow a_3;\ e : t_r}$$

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

- If $A = a_1$, then $t_A = \mathbf{data}$.
- If $A = a_2$, then $t_A = \mathbf{offset}$.
- If $A = a_3$, then $t_A = t_1$.

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-SetSub-Array.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : u\ \mathbf{array} \quad \widehat{\Gamma_r} \vdash a_2 : \mathbf{offset} \quad \widehat{\Gamma_r} \vdash a_3 : u \quad \widehat{\Gamma_r} \vdash e : t_r}{\widehat{\Gamma_r} \vdash a_1[a_2] \colon u \leftarrow a_3;\ e : t_r}$$

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

- If $A = a_1$, then $t_A = u\ \mathbf{array}$.
- If $A = a_2$, then $t_A = \mathbf{offset}$.
- If $A = a_3$, then $t_A = u$.

We can infer from the Atom typing Lemma 6.4.1 that there is some type t_a for which $\Gamma_r \vdash a : t_a$.

- Suppose the rule is Ty-SetSub-Tuple.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \widehat{\Gamma_r} \vdash a_3 : u \quad \widehat{\Gamma_r} \vdash e : t_r}{\widehat{\Gamma_r} \vdash a_1[\mathbf{offset}_c^{\mathbf{const}}(j)] \colon u \leftarrow a_3;\ e : t_r}$$

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

- If $A = a_1$, then $t_A = \langle u_0, \ldots, u_{i-1}, t_1, u_{i+1}, \ldots, u_{n-1} \rangle_c$.
- If $A = a_2$, then $t_A = t_1$.
- The case $A = \mathbf{offset}_c^{\mathbf{const}}(i)$ 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 $\Gamma_r \vdash a : t_a$.

- Suppose the rule is Ty-SetSub-Union.

$$\frac{\begin{array}{l} \widehat{\Gamma_r} \vdash a_1 : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) \\ \widehat{\Gamma_r} \vdash tv[t'_1, \ldots, t'_m] = \left(\lceil \vec{u}_i \rceil_0^{j-1} + \left\langle \lceil u'_i \rceil_0^{k-1}, u, \lceil u'_i \rceil_{k+1}^{l-1} \right\rangle + \lceil \vec{u}_i \rceil_{j+1}^{n-1} \right) : \omega_{union[n]} \\ \widehat{\Gamma_r} \vdash a_3 : u \\ \widehat{\Gamma_r} \vdash e : t_r \end{array}}{\widehat{\Gamma_r} \vdash a_1[\mathbf{offset}_{\mathbf{normal}}^{\mathbf{const}}(k)] \colon u \leftarrow a_3;\ e : t_r}$$

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

- If $A = a_1$, then $t_A = \mathbf{union}(tv[t'_1, \ldots, t'_k], \{j\})$.
- If $A = a_3$, then $t_A = t_1$.
- The case $A = \mathbf{offset}_{\mathbf{normal}}^{\mathbf{const}}(i)$ 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 $\Gamma_r \vdash a : t_a$.

- Suppose the rule is TY-SETSUB-FRAME.

$$\frac{\begin{array}{l} r_{\mathit{off}} \in \mathbb{Z}_{32}^{\mathbf{signed}} \\ \widehat{\Gamma_r} \vdash a_1 : \mathbf{frame}(tv[t_1, \ldots, t_m]) \\ \widehat{\Gamma_r} \vdash tv[t_1, \ldots, t_m] = \left\{ l^f : \{ l^r : u; \cdots \} ; \cdots \right\} : \omega_{\mathit{frame}} \\ \widehat{\Gamma_r} \vdash a_3 : u \\ \widehat{\Gamma_r} \vdash e : t_r \end{array}}{\widehat{\Gamma_r} \vdash a_1[\mathbf{label}(tv, l^f, l^r, r_{\mathit{off}})] \colon u \leftarrow a_3;\ e : t_r}$$

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

  - If $A = a_1$, then $t_A = \mathbf{frame}(tv[t'_1, \ldots, t'_m])$.
  - If $A = a_3$, then $t_A = t_1$.
  - The case $A = \mathbf{label}(tv, l^f, l^{sf}, i_{\mathit{off}})$ 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 $E = \mathbf{global}\, l \colon t \leftarrow A;\ e$. The proof for type judgment must invoke the rule TY-SETGLOBAL.

$$\frac{\widehat{\Gamma_r} \vdash l : u \quad \widehat{\Gamma_r} \vdash A : u \quad \widehat{\Gamma_r} \vdash e : t_r}{\widehat{\Gamma_r} \vdash \mathbf{global}\, l \colon u \leftarrow A;\ e : t_r}$$

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 $A = unop\, A'$. The proof $\Gamma \vdash A[a_r] : t$ must use the rule TY-ATOMUNOP, where $t = \mathbf{res}(unop)$.

$$\frac{\widehat{\Gamma} \vdash A'[a_r] : \mathbf{arg}(unop) \quad \widehat{\Gamma} \vdash \mathbf{res}(unop) : \Omega}{\widehat{\Gamma} \vdash unop\, A'[a_r] : \mathbf{res}(unop)}$$

By induction, we know $\Gamma \vdash A'[a_c] : \mathbf{arg}(unop)$, from which we can infer that $\Gamma \vdash unop\, A'[a_c] : \mathbf{res}(unop)$. △

Binary operations Suppose $A = A'\ binop\ a$. The proof $\Gamma \vdash A[a_r] : t$ must use the rule TY-ATOMBINOP.

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

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

This concludes the proof of the ATOM SUBSTITUTION Lemma 6.4.5. □

Lemma 6.4.6 SPECIALCALL SUBSTITUTION *Assume all of the following:*

- $\Gamma \vdash S[a_r] : \mathbf{special}\, t$
- $\Gamma \vdash a_r : u$,
- $\Gamma \vdash a_c : u$

Then $\Gamma \vdash S[a_c] : \mathbf{special}\, t$.

We prove this by structural induction on $S[a_r]$.

Migrate Suppose S is one of the following.

$$\begin{array}{rcl} S & ::= & \mathbf{migrate}\ [i, A, a_o]\ a_{fun}(a_1, \ldots, a_n) \\ & | & \mathbf{migrate}\ [i, h_p, A]\ a_{fun}(a_1, \ldots, a_n) \\ & | & \mathbf{migrate}\ [i, h_p, h_o]\ A(a_1, \ldots, a_n) \\ & | & \mathbf{migrate}\ [i, h_p, h_o]\ h_{fun}(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \end{array}$$

The type judgment must use the rule TY-SYSMIGRATE.

$$\frac{\begin{array}{cc} & \widehat{\Gamma} \vdash a_{ptr} : \mathbf{data} \\ j \in \mathbb{Z}_{31} & \widehat{\Gamma} \vdash a_{off} : \mathbb{Z}_{32}^{\mathbf{signed}} \\ \widehat{\Gamma} \vdash \lceil a_i : t_i \rceil_1^n & \widehat{\Gamma} \vdash a_f : (t_1, \ldots, t_n) \rightarrow \mathbf{enum}_0 \end{array}}{\widehat{\Gamma} \vdash \mathbf{migrate}\ [j, a_{ptr}, a_{off}]\ a_f(a_1, \ldots, a_n) : \mathbf{special}\, \mathbf{enum}_0}$$

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

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

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

Atomic Suppose S is one of the following.

$$\begin{array}{rcl} S & ::= & \mathbf{atomic}\ A(a_{const}, a_1, \ldots, a_n) \\ & | & \mathbf{atomic}\ v(A, a_1, \ldots, a_n) \\ & | & \mathbf{atomic}\ v(h_{const}, h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \end{array}$$

The type judgment must use the rule Ty-Atomic.

$$\frac{\widehat{\Gamma} \vdash a_{const} : \mathbb{Z}_{32}^{\mathbf{signed}} \quad \widehat{\Gamma} \vdash \lceil a_i : t_i \rceil_1^n \quad \widehat{\Gamma} \vdash a_f : (\mathbb{Z}_{32}^{\mathbf{signed}}, t_1, \ldots, t_n) \to \mathbf{enum}_0}{\widehat{\Gamma} \vdash \mathbf{atomic}\ a_f(a_{const}, a_1, \ldots, a_n) : \mathbf{special}\, \mathbf{enum}_0}$$

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

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

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

Atomic Rollback Suppose S is one of the following.

$$\begin{array}{rcl} S & ::= & \mathbf{rollback}\ [A, a_{const}] \\ & | & \mathbf{rollback}\ [h_{level}, A] \end{array}$$

The type judgment for S must use the rule Ty-AtomicRollback.

$$\frac{\widehat{\Gamma} \vdash a_{level} : \mathbb{Z}_{32}^{\mathbf{signed}} \quad \widehat{\Gamma} \vdash a_{const} : \mathbb{Z}_{32}^{\mathbf{signed}}}{\widehat{\Gamma} \vdash \mathbf{rollback}\ [a_{level}, a_{const}] : \mathbf{special}\, \mathbf{enum}_0}$$

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

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

Atomic Commit Suppose S is one of the following forms.

$$\begin{array}{rcl} S & ::= & \mathbf{commit}\ [A]\ a_{fun}(a_1, \ldots, a_n) \\ & | & \mathbf{commit}\ [h_{level}]\ A(a_1, \ldots, a_n) \\ & | & \mathbf{commit}\ [h_{level}]\ v(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \end{array}$$

The type judgment for S must use the rule TY-ATOMICCOMMIT.

$$\frac{\widehat{\Gamma} \vdash a_{level} : \mathbb{Z}_{32}^{\mathbf{signed}} \quad \widehat{\Gamma} \vdash \lceil a_i : t_i \rceil_1^n \quad \widehat{\Gamma} \vdash a_f : (t_1, \ldots, t_n) \to \mathbf{enum}_0}{\widehat{\Gamma} \vdash \mathbf{commit}\ [a_{level}]\ a_f(a_1, \ldots, a_n) : \mathbf{special}\,\mathbf{enum}_0}$$

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

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

From the ATOM SUBSTITUTION Lemma 6.4.5, we can conclude that $\Gamma \vdash A[a_c] : t_A$, from which we can infer that $\Gamma \vdash S[a_c] : \mathbf{special}\,t$. $\triangle$

This concludes the proof of the SPECIALCALL SUBSTITUTION Lemma 6.4.6. $\square$

Lemma 6.4.7 ALLOCATION SUBSTITUTION *Assume all of the following.*

- $\Gamma \vdash L[a_r] : \mathbf{alloc}\,t$,
- $\Gamma \vdash a_r : u$,
- $\Gamma \vdash a_c : u$.

Then $\Gamma \vdash L[a_c] : \mathbf{alloc}\,t$.

We prove this by structural induction on L.

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

$$\frac{\widehat{\Gamma} \vdash \lceil a_i : u_i \rceil_1^n \quad \widehat{\Gamma} \vdash \langle u_1, \ldots, u_n \rangle_c : \omega}{\widehat{\Gamma} \vdash \langle a_1, \ldots, a_n \rangle_c : \mathbf{alloc}\,\langle u_1, \ldots, u_n \rangle_c}$$

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 $\Gamma \vdash A[a_c] : u_i$, from which we can infer that $\Gamma \vdash L[a_c] : \mathbf{alloc}\,t$. $\triangle$

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

$$\frac{\widehat{\Gamma} \vdash tv_j(a_1, \ldots, a_k) : \mathbf{union}(tv[t_1, \ldots, t_m], \{j\})}{\widehat{\Gamma} \vdash \mathbf{union}(tv[a_1, \ldots, a_k], j) : \mathbf{alloc}\,\mathbf{union}(tv[t_1, \ldots, t_m], \{j\})}$$

The proof of the premise must use the TY-STOREUNION rule.

$$\frac{\begin{array}{l}\widehat{\Gamma}, \lceil \alpha_i : \omega = t'_i \rceil_1^m \vdash \lceil a_i : u_i \rceil_1^k \\ \widehat{\Gamma} \vdash \textbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) : \Omega \\ \widehat{\Gamma} \vdash tv[t'_1, \ldots, t'_m] = \vec{t}_0 + \cdots + \vec{t}_{j-1} + \langle u_1, \ldots, u_k \rangle + \vec{t}_{j+1} + \cdots + \vec{t}_{n-1} : \omega_{union[n]}\end{array}}{\widehat{\Gamma} \vdash tv_j(a_1, \ldots, a_k) : \textbf{union}(tv[t'_1, \ldots, t'_m], \{j\})}$$

From the premise $\Gamma, \lceil \alpha_i : \omega = t'_i \rceil_1^m \vdash a_i : u_i$, we can infer that $\Gamma, \lceil \alpha_i : \omega = t'_i \rceil_1^m \vdash A[a_r] : u_i$. From the Atom Substitution Lemma 6.4.5, we can conclude that $\Gamma, \lceil \alpha_i : \omega = t'_i \rceil_1^m \vdash A[a_c] : u_i$, from which we can infer that $\Gamma \vdash L[a_c] : \textbf{alloc}\, t$. △

Alloc array Suppose L is one of the following.

$$\begin{array}{rcl} L & ::= & \textbf{array}(A, a_{init}) : t \\ & | & \textbf{array}(h_{size}, A) : t \end{array}$$

The type judgment must use the rule Ty-Alloc-Array.

$$\frac{\widehat{\Gamma} \vdash a_{size} : \textbf{offset} \quad \widehat{\Gamma} \vdash a_{init} : u}{\widehat{\Gamma} \vdash \textbf{array}(a_{size}, a_{init}) : \textbf{alloc}\, u\ \textbf{array}}$$

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

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

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

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

$$\frac{\widehat{\Gamma} \vdash A[a_r] : \textbf{offset}}{\widehat{\Gamma} \vdash \textbf{malloc}(A[a_r]) : \textbf{alloc}\,\textbf{data}}$$

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

This concludes the proof of 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 $E = \mathbf{let}\, v\colon t_1 = A \;\mathbf{in}\; e$. The typing proof for E must use the rule TY-LETATOM.

$$\frac{\widehat{\Gamma} \vdash A[a_r] : u \quad \widehat{\Gamma}, v\colon u \vdash e : t}{\widehat{\Gamma} \vdash \mathbf{let}\, v\colon u = A[a_r] \;\mathbf{in}\; e : t}$$

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 $\Gamma \vdash \mathbf{let}\, v\colon t_1 = A[a_c] \;\mathbf{in}\; e : t$. △

LetExt Suppose $E = \mathbf{let}\, v_1\colon t_1 = (\text{“}s\text{”} : t_2)(h_1, \ldots, h_{i-1}, A, a_{i+1}, \ldots, a_n) \;\mathbf{in}\; e$. The proof of typing for $E[a_r]$ must end in the TY-LETEXT rule

$$\frac{\widehat{\Gamma} \vdash \lceil a_i : u_i \rceil_1^n \quad \widehat{\Gamma}, v\colon u \vdash e : t \quad \widehat{\Gamma} \vdash [\![\text{“s”}]\!] : (u_1, \ldots, u_n) \to u}{\widehat{\Gamma} \vdash \mathbf{let}\, v\colon u = (\text{“}s\text{”} : (u_1, \ldots, u_n) \to u)(a_1, \ldots, a_n) \;\mathbf{in}\; e : t}$$

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 $\Gamma \vdash E[a_c] : t$. △

TailCall Suppose $E = A(a_1, \ldots, a_n) : t$. The type proof must use the rule for TY-TAILCALL.

$$\frac{\widehat{\Gamma} \vdash A[a_r] : (u_1, \ldots, u_n) \to \mathbf{enum}_0 \quad \widehat{\Gamma} \vdash \lceil a_i : u_i \rceil_1^n}{\widehat{\Gamma} \vdash A[a_r](a_1, \ldots, a_n) : \mathbf{enum}_0}$$

From the premise $\Gamma \vdash A[a_r] : (t_1, \ldots, t_n) \to t$ we know $\Gamma \vdash A[a_r] : (t_1, \ldots, t_n) \to t$. From the ATOM SUBSTITUTION Lemma 6.4.5 we know $\Gamma \vdash A[a_c] : (t_1, \ldots, t_n) \to t$, 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 $\Gamma \vdash E[a_c] : t$. △

SpecialCall Suppose $E = \mathbf{special}\, S$. The type proof must use the rule for TY-SPECIAL-CALL.

$$\frac{\widehat{\Gamma} \vdash S[a_r] : \mathbf{special}\, t}{\widehat{\Gamma} \vdash \mathbf{special}\, S[a_r] : t}$$

From the premise $S[a_r] : \mathbf{special}\, t$ and the SPECIALCALL SUBSTITUTION Lemma 6.4.6, we know $S[a_c] : \mathbf{special}\, t$, from with we infer that $\Gamma \vdash E[a_c] : t$. △

Match Suppose $E = \mathbf{match}\, A \;\mathbf{with}\; \lceil s_i \mapsto e_i \rceil_1^n$. The typing proof must use one of the typing inferences for match expressions. Suppose the proof ends with the rule for TY-MATCH-INT.

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{31} \quad \widehat{\Gamma} \vdash A[a_r] : \mathbb{Z}_{31} \quad \widehat{\Gamma} \vdash \lceil e_i : t \rceil_1^n}{\widehat{\Gamma} \vdash \mathbf{match}\, A[a_r] \;\mathbf{with}\; \lceil s_i \mapsto e_i \rceil_1^n : t}$$

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. △

Alloc Suppose $E = \textbf{let}\, v = L \textbf{ in } e$. The typing proof must use the TY-ALLOC rule.

$$\frac{\widehat{\Gamma} \vdash L[a_r] : \textbf{alloc}\, u \quad \widehat{\Gamma}, v\colon u \vdash e : t}{\widehat{\Gamma} \vdash \textbf{let}\, v = L[a_r] \textbf{ in } e : t}$$

From the premise $\Gamma \vdash L[a_r] : \textbf{alloc}\, u$ and the ALLOCATION SUBSTITUTION Lemma 6.4.7, we can conclude $\Gamma \vdash L[a_c] : \textbf{alloc}\, u$, and infer that $\Gamma \vdash E[a_c] : t$. △

LetSubscript Suppose E is one of the following.

$$\begin{array}{rcl} E & ::= & \textbf{let}\, v\colon t = A[a_2] \textbf{ in } e \\ & | & \textbf{let}\, v\colon t = h[A] \textbf{ in } e \end{array}$$

The type judgment must end with one of the rules TY-LETSUB-RAWDATA, TY-LETSUB-ARRAY, TY-LETSUB-TUPLE, TY-LETSUB-UNION, or TY-LETSUB-FRAME. The cases are similar; we illustrate with the case for arrays.

$$\frac{\widehat{\Gamma} \vdash a_1 : u \textbf{ array} \quad \widehat{\Gamma} \vdash a_2 : \textbf{offset} \quad \widehat{\Gamma}, v\colon u \vdash e : t}{\widehat{\Gamma} \vdash \textbf{let}\, v\colon u = a_1[a_2] \textbf{ in } e : t}$$

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

- If $A[a_r] = a_1$, then $t_A = u$ **array**.
- If $A[a_r] = a_2$, then $t_A =$ **offset**.

From the ATOM SUBSTITUTION Lemma 6.4.5, we know $\Gamma \vdash A[a_c] : t_A$, and we can infer that $\Gamma \vdash E[a_c] : t$. △

SetSubscript Suppose E is one of the following.

$$\begin{array}{rcl} E & ::= & A[a_2]\colon t \leftarrow a_3;\ e \\ & | & v[A]\colon t \leftarrow a_3;\ e \\ & | & v[h]\colon t \leftarrow A;\ e \end{array}$$

The type judgment must end with one of the typing rules TY-SETSUB-RAWDATA, TY-SETSUB-ARRAY, TY-SETSUB-TUPLE, TY-SETSUB-UNION, or TY-SETSUB-FRAME. The cases are similar; we illustrate with the case for arrays.

$$\frac{\widehat{\Gamma} \vdash a_1 : u \textbf{ array} \quad \widehat{\Gamma} \vdash a_2 : \textbf{offset} \quad \widehat{\Gamma} \vdash a_3 : u \quad \widehat{\Gamma} \vdash e : t}{\widehat{\Gamma} \vdash a_1[a_2]\colon u \leftarrow a_3;\ e : t}$$

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

- If $A[a_r] = a_1$, then $t_A = u$ **array**.
- If $A[a_r] = a_2$, then $t_A =$ **offset**.
- If $A[a_r] = a_3$, then $t_A = u$.

From the ATOM SUBSTITUTION Lemma 6.4.5, we know $\Gamma \vdash A[a_c] : t_A$, and we can infer that $\Gamma \vdash E[a_c] : t$. △

SetGlobal Suppose $E = \textbf{global}\, l\colon t \leftarrow A;\ e$. The proof of typing must end with TY-SETGLOBAL.

$$\frac{\widehat{\Gamma} \vdash l : u \quad \widehat{\Gamma} \vdash A[a_r] : u \quad \widehat{\Gamma} \vdash e : t}{\widehat{\Gamma} \vdash \textbf{global}\, l\colon u \leftarrow A[a_r];\ e : t}$$

From the premise $A[a_r] : u$ and the ATOM SUBSTITUTION Lemma 6.4.5, we know $\Gamma \vdash A[a_c] : u$, and we can infer that $\Gamma \vdash E[a_c] : t$. △

□

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.*

$$\frac{\Gamma, \alpha\colon \Omega = u_r \vdash t_r = t_r : \Omega}{\Gamma, \alpha\colon \Omega = u_r \vdash t_r[u_r/\alpha] = t_r : \Omega} \qquad \text{TYPE-SUBST}$$

We prove this by structural induction on the type t_r.

Constant types If t_r is any of $\mathbb{Z}_{31}$, $\textbf{enum}_i$, $\mathbb{Z}_{pre}^{sign}$, $\mathbb{R}_{fpre}$, $\textbf{data}$, $v.i$, or a type variable $\beta \neq \alpha$, the substitution has no effect, and the result follows trivially. △

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).

$$\frac{\Gamma \vdash u_r = u_r : \Omega \quad \Gamma \vdash \alpha = \alpha : \Omega}{(\Gamma', \alpha\colon \Omega = u_r)\ \textbf{as}\ \Gamma \vdash \alpha = u_r : \Omega}$$

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. △

Tuples Suppose $t_r = \langle t_1, \ldots, t_n \rangle_{tuple\_class}$. 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.

$$\frac{\widehat{\Gamma} \vdash \lceil t_i = t_i : \omega \rceil_1^n \quad \widehat{\Gamma} \vdash \diamond}{\widehat{\Gamma} \vdash \langle t_1, \ldots, t_n \rangle_{\textbf{normal}} = \langle t_1, \ldots, t_n \rangle_{\textbf{normal}} : \omega}$$

From the premise $\lceil t_i = t_i : \omega \rceil_1^n$, we can assume by induction that $\lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n$.

We can then re-apply the WF-ST-TyTuple-Normal.

$$\frac{\Gamma \vdash \lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n \quad \Gamma \vdash \diamond}{\Gamma \vdash \langle t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha] \rangle_{\textbf{normal}} = \langle t_1, \ldots, t_n \rangle_{\textbf{normal}} : \omega}$$

$\triangle$

Arrays Suppose $t_r = t$ **array**. The proof that t_r is well-formed must use the rule WF-ST-TyArray.

$$\frac{\widehat{\Gamma} \vdash t = t : \Omega}{\widehat{\Gamma} \vdash t \textbf{ array} = t \textbf{ array} : \omega}$$

From the premise $t = t : \omega$, we can assume by induction that $t[u_r/\alpha] = t : \omega$.
We can then re-apply the WF-ST-TyArray rule.

$$\frac{\Gamma \vdash t[u_r/\alpha] = t : \Omega}{\Gamma \vdash t[u_r/\alpha] \textbf{ array} = t \textbf{ array} : \omega}$$

$\triangle$

Unions Suppose $t = \textbf{union}(tv[t_1, \ldots, t_n], set_I)$. The proof that t_r is a well-formed type must use the rule WF-ST-TyUnion.

$$\frac{set_I \subseteq \{0 \ldots n-1\} \quad \widehat{\Gamma} \vdash \lceil t_i = t_i : \omega \rceil_1^m \quad \widehat{\Gamma} \vdash tv[t_1, \ldots, t_m] : \omega_{union[n]}}{\Gamma \vdash \textbf{union}(tv[t_1, \ldots, t_m], set_I) = \textbf{union}(tv[t_1, \ldots, t_m], set_I) : \omega}$$

From the premise $\lceil t_i = t_i : \omega \rceil_1^n$, we can assume by induction that $\lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n$.
We can then re-apply the WF-ST-TyUnion rule.

$$\frac{set_I \subseteq \{0 \ldots n-1\} \quad \Gamma \vdash \lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^m \quad \Gamma \vdash tv[t_1, \ldots, t_m] : \omega_{union[n]}}{\Gamma \vdash \textbf{union}(tv[t_1[u_r/\alpha], \ldots, t_m[u_r/\alpha]], set_I) = \textbf{union}(tv[t_1, \ldots, t_m], set_I) : \omega}$$

$\triangle$

Frames Suppose $t_r = \textbf{frame}(tv[t_1, \ldots, t_n])$. The proof that t_r is a well-formed type must use the rule WF-ST-TyFrame.

$$\frac{\widehat{\Gamma} \vdash \lceil t_i = t_i : \omega \rceil_1^m \quad \widehat{\Gamma} \vdash tv[t_1, \ldots, t_m] : \omega_{frame}}{\Gamma \vdash \textbf{frame}(tv[t_1, \ldots, t_m]) = \textbf{frame}(tv[t_1, \ldots, t_m]) : \omega}$$

From the premise $\lceil t_i = t_i : \omega \rceil_1^n$, we can assume by induction that $\lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n$.
We can then re-apply the WF-ST-TyFrame rule.

$$\frac{\Gamma \vdash \lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^m \quad \Gamma \vdash tv[t_1, \ldots, t_m] : \omega_{frame}}{\Gamma \vdash \textbf{frame}(tv[t_1[u_r/\alpha], \ldots, t_m[u_r/\alpha]]) = \textbf{frame}(tv[t_1, \ldots, t_m]) : \omega}$$

$\triangle$

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 TY-APPLY-1.

$$\frac{\widehat{\Gamma} \vdash \lceil t_i = t_i : \omega \rceil_1^m \quad \widehat{\Gamma} \vdash tv : \omega^m \to k_s}{\widehat{\Gamma} \vdash tv[t_1, \ldots, t_m] = tv[t_1, \ldots, t_m] : k_s}$$

From the premise $\lceil t_i = t_i : \omega \rceil_1^n$, we can assume by induction that $\lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n$. We can then re-apply the TY-APPLY-1 rule.

$$\frac{\Gamma \vdash \lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^m \quad \Gamma \vdash tv : \omega^m \to k_s}{\Gamma \vdash tv[t_1[u_r/\alpha], \ldots, t_m[u_r/\alpha]] = tv[t_1, \ldots, t_m] : k_s}$$

$\triangle$

Functions Suppose $t_r = (t_1, \ldots, t_n) \to u$. The proof that t_r is well-formed must end with the rule WF-ST-TYFUN-MONO.

$$\frac{\widehat{\Gamma} \vdash \lceil t_i = t_i : \Omega \rceil_1^n \quad \widehat{\Gamma} \vdash u = u : \Omega}{\widehat{\Gamma} \vdash ((t_1, \ldots, t_n) \to u) = ((t_1, \ldots, t_n) \to u) : \omega}$$

From the premises $\lceil t_i = t_i : \omega \rceil_1^n$, and $u = u : \Omega$, we can assume by induction that $\lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n$ and $u[u_r/\alpha] = u : \Omega$. We can then re-apply the WF-ST-TYFUN-MONO rule.

$$\frac{\Gamma \vdash \lceil t_i[u_r/\alpha] = t_i : \Omega \rceil_1^n \quad \Gamma \vdash u[u_r/\alpha] = u : \Omega}{\Gamma \vdash ((t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha]) \to u[u_r/\alpha]) = ((t_1, \ldots, t_n) \to u) : \omega}$$

$\triangle$

Universal quantification Suppose $t_r = \forall \alpha_1, \ldots, \alpha_n.t$. Then t_r must be a function type $(t_1, \ldots, t_n) \to u$, and the proof that t_r is well-formed must end with the rule WF-ST-TYFUN-POLY.

$$\frac{\widehat{\Gamma}, \lceil \alpha_i : \omega \rceil_1^m \vdash ((t_1, \ldots, t_n) \to u) = ((t_1, \ldots, t_n) \to u) : \omega}{\widehat{\Gamma} \vdash \forall \alpha_1, \ldots, \alpha_m.((t_1, \ldots, t_n) \to u) = \forall \alpha_1, \ldots, \alpha_m.((t_1, \ldots, t_n) \to u) : \omega}$$

From the premises $\lceil t_i = t_i : \omega \rceil_1^n$, and $u = u : \Omega$, we can assume by induction that $\lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n$ and $u[u_r/\alpha] = u : \Omega$. We can then re-apply the WF-ST-TYFUN-POLY rule.

$$\frac{\Gamma, \lceil \alpha_i : \omega \rceil_1^m \vdash ((t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha]) \to u[u_r/\alpha]) = ((t_1, \ldots, t_n) \to u) : \omega}{\Gamma \vdash \forall \alpha_1, \ldots, \alpha_m.((t_1[u_r/\alpha], \ldots, t_n[u_r/\alpha]) \to u[u_r/\alpha]) = \forall \alpha_1, \ldots, \alpha_m.((t_1, \ldots, t_n) \to u) : \omega}$$

$\triangle$

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 WF-ST-TYEXISTS.

$$\frac{\widehat{\Gamma}, \lceil \alpha_i : \omega \rceil_1^m \vdash t = t : \Omega}{\widehat{\Gamma} \vdash \exists \alpha_1, \ldots, \alpha_m . t = \exists \alpha_1, \ldots, \alpha_m . t : \omega}$$

From the premises $\lceil t_i = t_i : \omega \rceil_1^n$, and $u = u : \Omega$, we can assume by induction that $\lceil t_i[u_r/\alpha] = t_i : \omega \rceil_1^n$. We can then re-apply the WF-ST-TyExists rule.

$$\frac{\Gamma, \lceil \alpha_i : \omega \rceil_1^m \vdash t[u_r/\alpha] = t : \Omega}{\Gamma \vdash \exists \alpha_1, \ldots, \alpha_m . t[u_r/\alpha] = \exists \alpha_1, \ldots, \alpha_m . t : \omega}$$

△

This concludes the proof of the $\dfrac{\Gamma, \alpha : \Omega = u_r \vdash t_r = t_r : \Omega}{\Gamma, \alpha : \Omega = u_r \vdash t_r[u_r/\alpha] = t_r : \Omega}$. □

6.6 Fully-defined contexts

Lemma 6.6.1 Fully-defined contexts *Assume all of the following are true.*

- Γ_r *is fully defined.*
- $\Gamma_r \vdash e_r : \mathbf{enum}_0$
- *For each checkpoint* $\langle \Gamma', f(\diamond, a_1, \ldots, a_n) \rangle \in \mathcal{C}_r$*, the environment* Γ' *is fully-defined, and* $\Gamma', v_c : \mathbb{Z}_{32}^{\mathbf{signed}} \vdash f(v_c, a_1, \ldots, a_n) : \mathbf{enum}_0$.
- $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to (\Gamma_c \mid \mathcal{C}_c \mid e_c)$

Then Γ_c *is fully defined, and for each* $\langle \Gamma', f(\diamond, a_1, \ldots, a_n) \rangle \in \mathcal{C}_c$*, the environment* Γ' *is fully-defined.*

The proof is by a case analysis on the reduction rule. We will consider only those operational rules that modify Γ_r or $\mathcal{C}_r$, as the other cases are trivial.

Type application Suppose $e_r = E[\mathbf{ty\_apply}[u](f, u_1, \ldots, u_m)]$. The reduction rule is Red-Atom-Apply.

$$\begin{array}{l}((\Gamma_r', f : u' = \Lambda \alpha_1, \ldots, \alpha_m . b) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid E[\mathbf{ty\_apply}[u](f, u_1, \ldots, u_m)]) \to \\ \quad (\Gamma_r, \lceil \alpha_i : \omega = u_i \rceil_1^m, g : u = b \mid \mathcal{C}_r \mid E[g])\end{array}$$

This rule introduces the new definitions $\lceil \alpha_i : \omega = u_i \rceil_1^m$ and $g : t = b$. △

Type abstraction Suppose $e_r = E[\mathbf{ty\_pack}[u](v_1, u_1, \ldots, u_m)]$. The operational rule is Red-Atom-Pack.

$$\begin{array}{c}((\Gamma_r', v_1 : u' = b) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid E[\mathbf{ty\_pack}[u](v_1, u_0, \ldots, u_{m-1})]) \to \\ (\Gamma_r, v_2 : u = \mathbf{ty\_pack}[u](v_1, u_0, \ldots, u_{m-1}) \mid \mathcal{C}_r \mid E[v_2])\end{array}$$

This rule introduces the new definition $v_2 : u = \mathbf{ty\_pack}[u](v_1, u_1, \ldots, u_m)$. △

Type unpacking Suppose $e_r = E[\mathbf{ty\_unpack}(v_1)]$. The operational rule is RED-ATOM-UNPACK.

$$\begin{array}{l}\Big((\Gamma'_r, v_1 \colon u = \mathbf{ty\_pack}[\exists\lceil\alpha_i\rceil_0^{m-1}.t](v, \lceil u_i\rceil_0^{m-1})) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid E[\mathbf{ty\_unpack}(v_1)]\Big) \to \\ \quad \Big(\Gamma_r, \lceil\alpha_i \colon \omega = u_i\rceil_0^{m-1}, v_2 \colon t = v \mid \mathcal{C}_r \mid E[v_2]\Big)\end{array}$$

The rule introduces the new definitions $\lceil\alpha_i \colon \omega = u_i\rceil_0^{m-1}$ and $v_2 \colon t = v$. △

LetAtom Suppose $e_r = \mathbf{let}\, v \colon u = h \textbf{ in } e$. The operational rule is RED-LETATOM.

$$(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{let}\, v \colon u = h \textbf{ in } e) \to (\Gamma_r, v \colon u = h \mid \mathcal{C}_r \mid e)$$

The rule introduces the new definition $v \colon u = h$. △

LetExt Suppose $e_r = \mathbf{let}\, v \colon t = (\text{“}s\text{”} : (u_1, \ldots, u_n) \to t)(h_1, \ldots, h_n) \textbf{ in } e$. The operational rule is RED-LETEXT.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{let}\, v \colon t = (\text{“}s\text{”} : (u_1, \ldots, u_n) \to t)(h_1, \ldots, h_n) \textbf{ in } e) \to \\ \quad (\Gamma_r, v \colon t = [\![\text{“s”}]\!](h_1, \ldots, h_n) \mid \mathcal{C}_r \mid e)\end{array}$$

The rule introduces the new definition $v \colon t = [\![\text{“s”}]\!](h_1, \ldots, h_n)$. △

TailCall Suppose $e_r = v(h_1, \ldots, h_n)$. The operational rule is RED-TAILCALL.

$$\begin{array}{l}((\Gamma'_r, v \colon (u_1, \ldots, u_n) \to t = \lambda v_1, \ldots, v_n.e) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid v(h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r, \lceil v_i \colon u_i = h_i\rceil_1^n \mid \mathcal{C}_r \mid e)\end{array}$$

The rule introduces the new definitions $\lceil v_i \colon u_i = h_i\rceil_1^n$. △

Special-calls There are several special-calls.

- For $e_r = \mathbf{special\,migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)$, the operational rule is RED-SYSMIGRATE.

  $$(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{special\,migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)) \to (\Gamma_r \mid \mathcal{C}_r \mid f(h_1, \ldots, h_n))$$

  The contexts are not modified.

- For $e_r = \mathbf{special\,atomic}\ f(h_{const}, h_1, \ldots, h_n)$, the operational rule is RED-ATOMIC.

  $$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{special\,atomic}\ f(h_{const}, h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r \mid \langle\Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle\,; \mathcal{C}_r \mid f(h_{const}, h_1, \ldots, h_n))\end{array}$$

  By assumption the context Γ_r is fully-defined.

- For $e_r = \mathbf{special\,rollback}\ [i, j]$, if $i \neq 0$, the operational rule is RED-ATOMIC-ROLLBACK-1.

  $$\begin{array}{l}(\Gamma'_r \mid C_m; \ldots; C_i = \langle\Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle\,; \ldots; C_1 \mid \mathbf{special\,rollback}\ [i, j]) \to \\ \quad (\Gamma_r \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, h_1, \ldots, h_n)) \\ \quad \mathbf{when}\, i \in \{1 \ldots m\}\end{array}$$

  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.

$$\begin{array}{l}(\Gamma_r' \mid C_m = \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n) \rangle \,; \ldots ; C_1 \mid \textbf{special rollback}\ [0, j]) \rightarrow \\ \quad (\Gamma_r \mid C_m; \ldots ; C_1 \mid f(j, h_1, \ldots, h_n))\end{array}$$

- For $e_r = \textbf{special commit}\ [i]\ f(h_1, \ldots, h_n)$, the operational rule is either RED-ATOMIC-COMMIT-1 or RED-ATOMIC-COMMIT-2.

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots ; C_1 \mid \textbf{special commit}\ [i]\ f(h_1, \ldots, h_n)) \rightarrow \\ \quad (\Gamma_r \mid C_m; \ldots ; C_{i+1}; C_{i-1}; \ldots ; C_1 \mid f(h_1, \ldots, h_n)) \\ \quad \textbf{when}\, i \in \{1 \ldots m\}\end{array}$$

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots ; C_1 \mid \textbf{special commit}\ [0]\ f(h_1, \ldots, h_n)) \rightarrow \\ \quad (\Gamma_r \mid C_{m-1}; \ldots ; C_1 \mid f(h_1, \ldots, h_n))\end{array}$$

In both cases, a checkpoint is removed from the checkpoint enviromnent $\mathcal{C}_r$, but the remaining checkpoints are not modified.

$\triangle$

Allocation There are several cases here for each of the allocation types.

- For $e_r = \textbf{let}\, v = \langle h_1, \ldots, h_n \rangle_c : u\ \textbf{in}\ e$, the operational rule is RED-ALLOC-TUPLE.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \langle h_1, \ldots, h_n \rangle_c : u\ \textbf{in}\ e) \rightarrow (\Gamma_r, v \colon u = \langle h_1, \ldots, h_n \rangle \mid \mathcal{C}_r \mid e)$$

The rule introduces the new definition $v \colon u = \langle h_1, \ldots, h_n \rangle$.

- For $e_r = \textbf{let}\, v = \textbf{union}(tv[h_1, \ldots, h_n], j) : t\ \textbf{in}\ e$, the operational rule is RED-ALLOC-UNION.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \textbf{union}(tv[h_1, \ldots, h_n], j) : t\ \textbf{in}\ e) \rightarrow (\Gamma_r, v \colon t = tv_j(h_1, \ldots, h_n) \mid \mathcal{C}_r \mid e)$$

The rule introduces the new definition $v \colon t = tv_j(h_1, \ldots, h_n)$.

- For $e_r = \textbf{let}\, v = \textbf{array}(i_{size}, h_{init}) : t\ \textbf{in}\ e$, the operational rule is RED-ALLOC-ARRAY.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \textbf{array}(i_{size}, h_{in}) : t\ \textbf{in}\ e) \rightarrow \left(\Gamma_r, v \colon t = \left\langle h_{in}^1, \ldots, h_{in}^{i_{size}} \right\rangle \mid \mathcal{C}_r \mid e\right) \\ \quad \textbf{when}\, i_{size} \geq 0\end{array}$$

The rule introduces the new definition $v \colon t = \left\langle h_{init}^1, \ldots, h_{init}^{i_{size}} \right\rangle$.

- For $e_r = \textbf{let}\, v = \textbf{malloc}(i_{size}) : t\ \textbf{in}\ e$, the operational rule is RED-ALLOC-MALLOC.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \textbf{malloc}(i_{size}) : t\ \textbf{in}\ e) \rightarrow (\Gamma_r, v \colon t = \langle \mathrm{C} \rangle \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\, i_{size} \geq 0\end{array}$$

The rule introduces the new definition $v \colon t = \langle \mathrm{C} \rangle$.

- For $e_r = \textbf{let}\, v = \textbf{frame}(tv[t_1, \ldots, t_m])\ \textbf{in}\ e$, the operation rule is RED-ALLOC-FRAME.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \textbf{frame}(tv[t_1, \ldots, t_m])\ \textbf{in}\ e) \rightarrow \\ \quad (\Gamma_r, v \colon \textbf{frame}(tv[t_1, \ldots, t_m]) = \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \mid \mathcal{C}_r \mid e)\end{array}$$

The rule introduces the new definition $v \colon \textbf{frame}(tv[t_1, \ldots, t_m]) = \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle$.

$\triangle$

Subscripting There are several subscript cases for the expression $e_r = \textbf{let}\, v_1 : t_1 = v_2[i]\ \textbf{in}\ e$.

- The operational rule RED-LETSUB-TUPLE inroduces the new definition $v_1 : u = h_j$.

$$\begin{array}{l}((\Gamma'_r, v_2 : u' = \langle h_0, \ldots, h_{n-1}\rangle)\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j]\ \textbf{in}\ e) \to \\ \quad (\Gamma_r, v_1 : u = h_j \mid \mathcal{C}_r \mid e)\end{array}$$

- The operational rule RED-LETSUB-ARRAY introduces the new definition $v_1 : u = h_j$.

$$\begin{array}{l}((\Gamma'_r, v_2 : u' = \langle h_0, \ldots, h_{n-1}\rangle)\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j]\ \textbf{in}\ e) \to \\ \quad (\Gamma_r, v_1 : u = h_j \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\, j \in \{0 \ldots n-1\}\end{array}$$

- The operational rule RED-LETSUB-UNION introduces the new definition $v_1 : u = h_k$.

$$\begin{array}{l}((\Gamma'_r, v_2 : u' = tv_j(h_0, \ldots, h_{n-1}))\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[k]\ \textbf{in}\ e) \to \\ \quad (\Gamma_r, v_1 : u = h_k \mid \mathcal{C}_r \mid e)\end{array}$$

- The operational rule RED-LETSUB-RAWDATA introduces the new definition $v_1 : u = h$.

$$\begin{array}{l}((\Gamma'_r, v_2 : u' = \langle \mathrm{C}\rangle)\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j]\ \textbf{in}\ e) \to \\ \quad (\Gamma_r, v_1 : u = h \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\,\textbf{runtime}(\Gamma_r \mid \langle \mathrm{C}\rangle\, [j] : u) = h\end{array}$$

- The operational rule RED-LETSUB-FRAME introduces the new definition $v_1 : u = h$.

$$\begin{array}{l}((\Gamma'_r, v_2 : u' = \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m])\rangle)\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j]\ \textbf{in}\ e) \to \\ \quad (\Gamma_r, v_1 : u = h \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\,\textbf{runtime}(\Gamma_r \mid \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m])\rangle\, [j] : u) = h\end{array}$$

$\triangle$

Subscript assignment There are several subscript assignment cases for the expression $v_1[i] : t_2 \leftarrow h;\ e$.

- RED-SETSUB-TUPLE

$$\begin{array}{l}((\Gamma'_r, v : u' = \langle h_0, \ldots, h_{n-1}\rangle)\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid v[j] : u \leftarrow h;\ e) \to \\ \quad (\Gamma'_r, v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1}\rangle \mid \mathcal{C}_r \mid e)\end{array}$$

- RED-SETSUB-ARRAY

$$\begin{array}{l}((\Gamma'_r, v : u' = \langle h_0, \ldots, h_{n-1}\rangle)\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid v[j] : u \leftarrow h;\ e) \to \\ \quad (\Gamma'_r, v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1}\rangle \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\, j \in \{0 \ldots n-1\}\end{array}$$

- RED-SETSUB-UNION

$$\begin{array}{l}(\Gamma_r, v : u' = tv_j(h_0, \ldots, h_{n-1}) \mid \mathcal{C}_r \mid v[k] : u \leftarrow h;\ e) \to \\ \quad (\Gamma_r, v : u' = tv_j(h_0, \ldots, h_{k-1}, h, h_{k+1}, \ldots, h_{n-1}) \mid \mathcal{C}_r \mid e)\end{array}$$

- RED-SETSUB-RAWDATA

$$\begin{array}{l}((\Gamma'_r, v\colon u' = \langle \text{C} \rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid v[j]\colon u \leftarrow h;\ e) \rightarrow \\ \quad (\Gamma'_r, v\colon u' = \langle \text{C}' \rangle \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when runtime}(\Gamma_r \mid \langle \text{C} \rangle\, [j] : u \leftarrow h) = \langle \text{C}' \rangle\end{array}$$

- RED-SETSUB-FRAME

$$\begin{array}{l}((\Gamma'_r, v\colon u' = \langle \text{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid v[j]\colon u \leftarrow h;\ e) \rightarrow \\ \quad (\Gamma'_r, v\colon u' = \langle \text{C}' : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when runtime}(\Gamma_r \mid \langle \text{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle\, [j] : u \leftarrow h) \\ \qquad = \langle \text{C}' : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle\end{array}$$

In each of the cases, the definition in the environment is modified, but remains a definition. $\triangle$

LetGlobal Suppose $e_r = \textbf{let}\, v\colon u = \textbf{global}\, l \textbf{ in } e$. The operational rule is RED-LETGLOBAL.

$$((\Gamma'_r, l\colon u = h) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v\colon u = \textbf{global}\, l \textbf{ in } e) \rightarrow (\Gamma_r, v\colon u = h \mid \mathcal{C}_r \mid e)$$

The rule adds the definition $v\colon u = h$ to the context. $\triangle$

SetGlobal Suppose $e_r = \textbf{global}\, l\colon u \leftarrow h;\ e$. The operational rule is RED-SETGLOBAL.

$$((\Gamma'_r, l\colon u = h') \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{global}\, l\colon u \leftarrow h;\ e) \rightarrow (\Gamma'_r, l\colon u = h \mid \mathcal{C}_r \mid e)$$

The definition $l\colon u' = h'$ 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* $v \in dom(\Gamma)$.

The only rule that can be used to derive a type for a variable is the TY-ATOMVAR rule.

$$\frac{\Gamma, v\colon t \vdash \diamond}{\Gamma, v\colon t \vdash v : t}$$

$\square$

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* $\mathbb{Z}_{31}$*,* $\textbf{enum}_n$*,* $\mathbb{Z}^{sign}_{pre}$*, and* $\mathbb{R}_{fpre}$*, 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.

$$\frac{i \in \mathbb{Z}_{31} \quad \Gamma \vdash \diamond}{\Gamma \vdash \mathbf{int}(i) : \mathbb{Z}_{31}}$$

- In the second case, the value is derived using the TY-ATOMVAR rule.

$$\frac{\Gamma, v\colon \mathbb{Z}_{31} \vdash \diamond}{\Gamma, v\colon \mathbb{Z}_{31} \vdash v : \mathbb{Z}_{31}}$$

  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 Γ, 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.

$$\frac{\Gamma, v\colon (u_1, \ldots, u_n) \to t \vdash \diamond}{\Gamma, v\colon (u_1, \ldots, u_n) \to t \vdash v : (u_1, \ldots, u_n) \to t}$$

That is, h must be a variable v. The only possible typing rule is the TY-STOREFUN-MONO rule.

$$\frac{\Gamma, \lceil v_i\colon u_i \rceil_1^n \vdash e : t}{\Gamma \vdash \lambda v_1, \ldots, v_n.e : (u_1, \ldots, u_n) \to t}$$

$\square$

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* $(\Gamma_r \mid \mathcal{C}_r \mid e_r)$ *is a valid program and* $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \rightarrow (\Gamma_c \mid \mathcal{C}_c \mid e_c)$*, then* $(\Gamma_c \mid \mathcal{C}_c \mid e_c)$ *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 $\mathcal{C}_r$ and the context Γ_r are fully-defined. The preservation proof amounts to showing that types are preserved during reduction.

The proof depends on the type judgment $\Gamma_r \vdash e_r : \mathbf{enum}_0$. 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 $\widehat{\Gamma_r}$ where some number of thinning rules have been applied.

Variable reduction Suppose the reduction uses the rule RED-ATOM-VAR.

$$(\Gamma_r, v : t_1 = h \mid \mathcal{C}_r \mid E[v]) \rightarrow (\Gamma_r, v : t_1 = h \mid \mathcal{C}_r \mid E[h])$$

By assumption $\Gamma_r, v : t_1 = h \vdash E[v] : \mathbf{enum}_0$. 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 $\Gamma_r, v : t_1 = h \vdash E[h] : \mathbf{enum}_0$. $\triangle$

Unary operations Suppose the reduction uses the RED-ATOM-UNOP rule.

$$(\Gamma_r \mid \mathcal{C}_r \mid E[\mathit{unop}\, i]) \rightarrow (\Gamma_r \mid \mathcal{C}_r \mid E[[\![\mathit{unop}]\!](i)])$$

From the assumption $\Gamma_r \vdash E[\mathit{unop}\, i] : \mathbf{enum}_0$ and the Expression typing Lemma 6.4.4, we know that the atom $\mathit{unop}\, i$ has some type t_a. The proof of typing must use the Ty-AtomUnop rule.

$$\frac{\widehat{\Gamma_r} \vdash i : \mathbf{arg}(\mathit{unop}) \quad \widehat{\Gamma_r} \vdash \mathbf{res}(\mathit{unop}) : \Omega}{\widehat{\Gamma_r} \vdash \mathit{unop}\, i : \mathbf{res}(\mathit{unop})}$$

That is, t_a is $\mathbf{res}(\mathit{unop})$. Since the builtin unary operators are well-typed (by assumption), the value $[\![\mathit{unop}]\!](i)$ also has type $\mathbf{res}(\mathit{unop})$. We can conclude from the Substitution Lemma 6.4.8 that $\Gamma_r \vdash E[[\![\mathit{unop}]\!](i)] : t$. △

Binary operations Suppose the reduction uses the Red-Atom-Binop rule.

$$(\Gamma_r \mid \mathcal{C}_r \mid E[i\ \mathit{binop}\ j]) \rightarrow (\Gamma_r \mid \mathcal{C}_r \mid E[[\![\mathit{binop}]\!](i,j)])$$

From the assumption $\Gamma_r \vdash E[i\ \mathit{binop}\ j] : \mathbf{enum}_0$ and the Expression typing Lemma 6.4.4, we know that the atom $i\ \mathit{binop}\ j$ has some type t_a. The proof of typing must use the Ty-AtomBinop rule.

$$\frac{\widehat{\Gamma_r} \vdash i : \mathbf{arg}_1(\mathit{binop}) \quad \widehat{\Gamma_r} \vdash j : \mathbf{arg}_2(\mathit{binop}) \quad \widehat{\Gamma_r} \vdash \mathbf{res}(\mathit{binop}) : \Omega}{\widehat{\Gamma_r} \vdash i\ \mathit{binop}\ j : \mathbf{res}(\mathit{binop})}$$

That is, t_a is $\mathbf{res}(\mathit{binop})$. Since the builtin binary operators are well-typed (by assumption), the value $[\![\mathit{binop}]\!](i,j)$ also has type $\mathbf{res}(\mathit{binop})$. We can conclude from the Substitution Lemma 6.4.8 that $\Gamma_r \vdash E[[\![\mathit{binop}]\!](i,j)] : \mathbf{enum}_0$. △

Type application Suppose the reduction uses the Red-Atom-Apply rule.

$$\begin{array}{l}((\Gamma'_r, f\colon u' = \Lambda\alpha_1,\ldots,\alpha_m.b)\ \mathbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid E[\mathbf{ty\_apply}[u](f,u_1,\ldots,u_m)]) \rightarrow \\ \quad (\Gamma_r, \lceil \alpha_i\colon \omega = u_i \rceil_1^m, g\colon u = b \mid \mathcal{C}_r \mid E[g])\end{array}$$

From the assumption $\Gamma_r \vdash E[\mathbf{ty\_apply}[u](v_1,u_1,\ldots,u_m)] : \mathbf{enum}_0$, and the Expression typing Lemma 6.4.4, we know that the atom $\mathbf{ty\_apply}[u](v_1,u_1,\ldots,u_m)$ has some type t_a. The proof of typing must use the Ty-AtomTyApply rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil u_i : \omega \rceil_1^m \quad \widehat{\Gamma_r} \vdash v_1 : \forall\alpha_1,\ldots,\alpha_m.t \quad \widehat{\Gamma_r} \vdash u = t[\lceil u_i/\alpha_i \rceil_1^m] : \omega}{\widehat{\Gamma_r} \vdash \mathbf{ty\_apply}[u](v_1,u_1,\ldots,u_m) : t[\lceil u_i/\alpha_i \rceil_1^m]}$$

That is, $t_a = u = t[\lceil u_i/\alpha_i \rceil_1^m]$. By the Type substitution Lemma 6.5.1, we can infer that

$$\Gamma_r, \lceil \alpha_i\colon \omega = u_i \rceil_1^m, v_2\colon t = b \vdash v_2 : t_a$$

We can conclude from the Substitution Lemma 6.4.8 that $\Gamma_r \vdash E[v_2] : \mathbf{enum}_0$. △

Type abstraction Suppose the reduction uses the Red-Atom-Pack rule.

$$\begin{array}{l}((\Gamma'_r, v_1 : u' = b)\ \mathbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid E[\mathbf{ty\_pack}[u](v_1,u_0,\ldots,u_{m-1})]) \rightarrow \\ \quad (\Gamma_r, v_2\colon u = \mathbf{ty\_pack}[u](v_1,u_0,\ldots,u_{m-1}) \mid \mathcal{C}_r \mid E[v_2])\end{array}$$

From the assumption $\Gamma_r \vdash E[\textbf{ty\_pack}[u](v_1, u_1, \ldots, u_m)] : \textbf{enum}_0$, and the EXPRESSION TYPING Lemma 6.4.4, we know that the atom $\textbf{ty\_pack}[u](v_1, u_1, \ldots, u_m)$ has some type t_a. The proof of typing must use the TY-ATOMTYPACK rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil u_i : \omega \rceil_1^m \quad \widehat{\Gamma_r} \vdash v_1 : t[\lceil u_i/\alpha_i \rceil_1^m] \quad \widehat{\Gamma_r} \vdash u = \exists \alpha_1, \ldots, \alpha_m.t : \omega}{\widehat{\Gamma_r} \vdash \textbf{ty\_pack}[u](v_1, u_1, \ldots, u_m) : \exists \alpha_1, \ldots, \alpha_m.t}$$

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 $\Gamma_r \vdash E[v_2] : \textbf{enum}_0$. △

Type unpacking Suppose the reduction uses the RED-ATOM-UNPACK rule.

$$\Big((\Gamma'_r, v_1 : u = \textbf{ty\_pack}[\exists \lceil \alpha_i \rceil_0^{m-1}.t](v, \lceil u_i \rceil_0^{m-1})) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid E[\textbf{ty\_unpack}(v_1)]\Big) \rightarrow$$
$$\Big(\Gamma_r, \lceil \alpha_i : \omega = u_i \rceil_0^{m-1}, v_2 : t = v \mid \mathcal{C}_r \mid E[v_2]\Big)$$

From the assumption $\Gamma_r \vdash E[\textbf{ty\_unpack}(v_1)] : t$, and the EXPRESSION TYPING Lemma 6.4.4, we know that the atom $\textbf{ty\_unpack}(v_1)$ has some type t_a. The proof of typing must use the TY-ATOMTYUNPACK rule.

$$\frac{\widehat{\Gamma_r} \vdash v_1 : \exists \alpha_0, \ldots, \alpha_{m-1}.t}{\widehat{\Gamma_r} \vdash \textbf{ty\_unpack}(v_1) : t[\lceil v_1.i/\alpha_i \rceil_0^{m-1}]}$$

That is, $t_a = t[\lceil v_1.i/\alpha_i \rceil_0^m]$. Since $v_1.i = u_i$ for each $i \in \{0 \ldots m-1\}$, we can infer from the TYPE SUBSTITUTION Lemma 6.5.1 that $\Gamma_r \vdash E[v_2] : \textbf{enum}_0$. △

LetAtom Suppose the reduction uses the rule RED-LETATOM.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v : u = h \textbf{ in } e) \rightarrow (\Gamma_r, v : u = h \mid \mathcal{C}_r \mid e)$$

The typing proof must use the rule TY-LETATOM.

$$\frac{\widehat{\Gamma_r} \vdash h : u \quad \widehat{\Gamma_r}, v : u \vdash e : \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{let}\, v : u = h \textbf{ in } e : \textbf{enum}_0}$$

Using the THIN-VAR-DEF rule and the premises, we can infer $\Gamma_r, v : t_1 = h \vdash e_r : \textbf{enum}_0$.

$$\frac{\Gamma_r, v : u \vdash h : u \quad \Gamma_r, v : u \vdash e : \textbf{enum}_0}{\Gamma_r, v : u = h \vdash e : \textbf{enum}_0}$$

△

LetExt Suppose the reduction uses the rule RED-LETEXT.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v : t = (\text{“}s\text{”} : (u_1, \ldots, u_n) \rightarrow t)(h_1, \ldots, h_n) \textbf{ in } e) \rightarrow$$
$$(\Gamma_r, v : t = [\![\text{“s”}]\!](h_1, \ldots, h_n) \mid \mathcal{C}_r \mid e)$$

The typing proof must use the rule TY-LETEXT.

$$\frac{\widehat{\Gamma_r} \vdash \lceil h_i : u_i \rceil_1^n \quad \widehat{\Gamma_r}, v\colon u \vdash e : \mathbf{enum}_0 \quad \widehat{\Gamma_r} \vdash [\![\text{“s”}]\!] : (u_1, \ldots, u_n) \to u}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v\colon u = (\text{“}s\text{”} : (u_1, \ldots, u_n) \to u)(h_1, \ldots, h_n)\ \mathbf{in}\ e : \mathbf{enum}_0}$$

From the premises $\widehat{\Gamma_r} \vdash \lceil h_i : u_i \rceil_1^n$ and $\widehat{\Gamma_r} \vdash [\![\text{“s”}]\!] : (u_1, \ldots, u_n) \to u$, we can infer that $\widehat{\Gamma_r} \vdash [\![\text{“s”}]\!](h_1, \ldots, h_n) : u$. Using the THIN-VAR-DEF rule, we can infer that $\Gamma_r, v\colon u = [\![\text{“s”}]\!](h_1, \ldots, h_n) \vdash e_r : \mathbf{enum}_0$.

$$\frac{\Gamma_r, v\colon u \vdash [\![\text{“s”}]\!](h_1, \ldots, h_n) : u \quad \Gamma_r, v\colon u \vdash e : \mathbf{enum}_0}{\Gamma_r, v\colon u = [\![\text{“s”}]\!](h_1, \ldots, h_n) \vdash e : \mathbf{enum}_0}$$

△

TailCall Suppose the reduction uses the rule RED-TAILCALL.

$$\begin{aligned}&((\Gamma_r', v\colon (u_1, \ldots, u_n) \to t = \lambda v_1, \ldots, v_n.e)\ \mathbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid v(h_1, \ldots, h_n)) \to \\ &\quad (\Gamma_r, \lceil v_i\colon u_i = h_i \rceil_1^n \mid \mathcal{C}_r \mid e)\end{aligned}$$

The proof of typing must use the rule TY-TAILCALL.

$$\frac{\widehat{\Gamma_r} \vdash f : (u_1, \ldots, u_n) \to \mathbf{enum}_0 \quad \widehat{\Gamma_r} \vdash \lceil h_i : u_i \rceil_1^n}{\widehat{\Gamma_r} \vdash f(h_1, \ldots, h_n) : \mathbf{enum}_0}$$

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.

$$\frac{\widehat{\Gamma_r, \lceil v_i\colon u_i \rceil_1^n} \vdash e : \mathbf{enum}_0}{\widehat{\Gamma_r} \vdash \lambda v_1, \ldots, v_n.e : (u_1, \ldots, u_n) \to \mathbf{enum}_0}$$

From the THIN-VAR-DEF rule, we can conclude that

$$\Gamma_r, \lceil v_i\colon u_i = h_i \rceil_1^n \vdash e : \mathbf{enum}_0.$$

△

SpecialCall There are four cases for reducing a special-call.

- Suppose the reduction uses the rule RED-SYSMIGRATE.

$$(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{special\,migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)) \to (\Gamma_r \mid \mathcal{C}_r \mid f(h_1, \ldots, h_n))$$

The proof of typing must use the rule TY-SYSMIGRATE.

$$\frac{\begin{array}{c}\widehat{\Gamma_r} \vdash h_{ptr} : \mathbf{data} \\ j \in \mathbb{Z}_{31} \qquad \widehat{\Gamma_r} \vdash h_{off} : \mathbb{Z}_{32}^{\mathbf{signed}} \\ \widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash f : (t_1, \ldots, t_n) \to \mathbf{enum}_0\end{array}}{\widehat{\Gamma_r} \vdash \mathbf{migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n) : \mathbf{special\,enum}_0}$$

This case is similar to a tail-call. From the premises $\widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n$ and $\widehat{\Gamma_r} \vdash f : (t_1, \ldots, t_n) \to \mathbf{enum}_0$, we can infer that $\Gamma_r \vdash f(h_1, \ldots, h_n) : \mathbf{enum}_0$.

- Suppose the reduction uses the rule RED-ATOMIC.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{special atomic } f(h_{const}, h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r \mid \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle \,; \mathcal{C}_r \mid f(h_{const}, h_1, \ldots, h_n))\end{array}$$

The proof of typing must use the rule TY-ATOMIC.

$$\frac{\widehat{\Gamma_r} \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash f : (\mathbb{Z}_{32}^{\textbf{signed}}, t_1, \ldots, t_n) \to \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{atomic } f(h_{const}, h_1, \ldots, h_n) : \textbf{special enum}_0}$$

From the premises $\widehat{\Gamma_r} \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}}$, $\widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n$, and $\widehat{\Gamma_r} \vdash f : (t_1, \ldots, t_n) \to \textbf{enum}_0$, we can infer that the checkpoint $\langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle$ is well-formed, and that $\Gamma_r \vdash f(h_1, \ldots, h_n) :$ $\textbf{enum}_0$

- Suppose the reduction uses the rule RED-ATOMIC-ROLLBACK-1.

$$\begin{array}{l}(\Gamma'_r \mid C_m; \ldots; C_i = \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle \,; \ldots; C_1 \mid \textbf{special rollback } [i, j]) \to \\ \quad (\Gamma_r \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, h_1, \ldots, h_n)) \\ \quad \textbf{when } i \in \{1 \ldots m\}\end{array}$$

By assumption, the checkpoint context $\mathcal{C}_r$ is well-formed, and $\Gamma_r \vdash f(j, h_1, \ldots, h_n) : \textbf{enum}_0$. 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.

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots; C_1 \mid \textbf{special commit } [i] \; f(h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) \\ \quad \textbf{when } i \in \{1 \ldots m\}\end{array}$$

The proof of typing must use the rule TY-ATOMICCOMMIT.

$$\frac{\widehat{\Gamma_r} \vdash i : \mathbb{Z}_{32}^{\textbf{signed}} \quad \widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash f : (t_1, \ldots, t_n) \to \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{commit } [i] \; f(h_1, \ldots, h_n) : \textbf{special enum}_0}$$

From the premises $\widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n$ and $\widehat{\Gamma_r} \vdash f : (t_1, \ldots, t_n) \to \textbf{enum}_0$, we can infer that $\Gamma_r \vdash$ $f(h_1, \ldots, h_n) : \textbf{enum}_0$. 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.

$$\begin{array}{l}\left(\Gamma_r \mid \mathcal{C}_r \mid \textbf{match } i \textbf{ with } \lceil s_j \mapsto e_j \rceil_{j=1}^n\right) \to (\Gamma_r \mid \mathcal{C}_r \mid e_k) \\ \quad \textbf{when } (i \in s_k) \wedge \forall j \in \{1, \ldots, k-1\}.(i \notin s_j)\end{array}$$

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.

$$\frac{\lceil s_j \in set_I \rceil_{j=1}^{n} \quad \bigcup_{j=1}^{n} s_j = \mathbb{Z}_{31} \quad \widehat{\Gamma_r} \vdash i : \mathbb{Z}_{31} \quad \widehat{\Gamma_r} \vdash \lceil e_j : \mathbf{enum}_0 \rceil_{j=1}^{n}}{\widehat{\Gamma_r} \vdash \mathbf{match}\, i \ \mathbf{with}\ \lceil s_j \mapsto e_j \rceil_{j=1}^{n} : \mathbf{enum}_0}$$

We can infer that $\Gamma_r \vdash e_k : t$ directly from the premise $\Gamma_r \vdash \lceil e_j : t \rceil_{j=1}^{n}$.

- Suppose the reduction uses the rule RED-MATCH-UNION.

$$\begin{aligned}&\left(\Gamma_r, v\colon \mathbf{union}(tv[t_1,\ldots,t_m], s) = tv_j(a_1,\ldots,a_n) \mid \mathcal{C}_r \mid \mathbf{match}\, v \ \mathbf{with}\ \lceil s_i \mapsto e_i \rceil_1^l\right) \rightarrow \\ &\quad (\Gamma_r, v\colon \mathbf{union}(tv[t_1,\ldots,t_m], s_k) = tv_j(a_1,\ldots,a_n) \mid \mathcal{C}_r \mid e_k) \\ &\quad \mathbf{when}\ (j \in s_k) \wedge \forall p \in \{1,\ldots,k-1\}.(i \notin s_p)\end{aligned}$$

The only possible typing rule for this expression is TY-MATCH-UNION.

$$\frac{\lceil s_j \in set_I \rceil_{j=1}^{n} \quad \{\} \neq s_{union} = \bigcup_{j=1}^{n} s_j \quad \left\lceil \widehat{\Gamma_r}, v\colon \mathbf{union}(tv[t_1,\ldots,t_m], s_j) \vdash e_j : \mathbf{enum}_0 \right\rceil_{j=1}^{n}}{\widehat{\Gamma_r}, v\colon \mathbf{union}(tv[t_1,\ldots,t_m], s_{union}) \vdash \mathbf{match}\, v \ \mathbf{with}\ \lceil s_j \mapsto e_j \rceil_{j=1}^{n} : \mathbf{enum}_0}$$

We can infer that $\Gamma_r, v\colon \mathbf{union}(tv[t_1,\ldots,t_m], s_k) = tv_i(a_1,\ldots,a_n) \vdash e_k : \mathbf{enum}_0$ directly from the premises.

$\triangle$

LetAlloc There are several allocation cases. In each of these cases, the type judgment uses the TY-ALLOC rule.

$$\frac{\widehat{\Gamma_r} \vdash alloc : \mathbf{alloc}\, u \quad \widehat{\Gamma_r}, v\colon u \vdash e : \mathbf{enum}_0}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v = alloc \ \mathbf{in}\ e : \mathbf{enum}_0}$$

- Suppose the reduction uses the rule RED-ALLOC-TUPLE.

$$(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{let}\, v = \langle h_1,\ldots,h_n \rangle_c : u \ \mathbf{in}\ e) \rightarrow (\Gamma_r, v\colon u = \langle h_1,\ldots,h_n \rangle \mid \mathcal{C}_r \mid e)$$

The corresponding type judgment is TY-ALLOC-TUPLE.

$$\frac{\widehat{\Gamma_r} \vdash \lceil h_i : u_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash \langle u_1,\ldots,u_n \rangle_c : \omega}{\widehat{\Gamma_r} \vdash \langle h_1,\ldots,h_n \rangle_c : \mathbf{alloc}\, \langle u_1,\ldots,u_n \rangle_c}$$

From the premises $\Gamma_r \vdash \lceil h_i : u_i \rceil_1^n$ we can infer that the $u = \langle u_1,\ldots,u_n \rangle_{tuple\_class}$ and the context $\Gamma_r, v\colon u = \langle h_1,\ldots,h_n \rangle$ is well-formed. From the premise $e : \mathbf{enum}_0$ we conclude that $\Gamma_r, v\colon u = \langle h_1,\ldots,h_n \rangle \vdash e : \mathbf{enum}_0$.

- Suppose the reduction uses the rule RED-ALLOC-ARRAY.

$$\begin{aligned}&(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{let}\, v = \mathbf{array}(i_{size}, h_{in}) : \mathbf{enum}_0 \ \mathbf{in}\ e) \rightarrow \left(\Gamma_r, v\colon \mathbf{enum}_0 = \left\langle h_{in}^1,\ldots,h_{in}^{i_{size}} \right\rangle \mid \mathcal{C}_r \mid e\right) \\ &\quad \mathbf{when}\, i_{size} \geq 0\end{aligned}$$

The corresponding type judgment is TY-ALLOC-ARRAY.

$$\frac{\widehat{\Gamma_r} \vdash i_{size} : \textbf{offset} \quad \widehat{\Gamma_r} \vdash h_{init} : u'}{\widehat{\Gamma_r} \vdash \textbf{array}(i_{size}, h_{init}) : \textbf{alloc}\, u'\ \textbf{array}}$$

That is, $u = u'$ **array**. From the premise $\Gamma_r \vdash h_{init} : u'$ we can infer that the context $\Gamma_r, v\colon u = \langle h_1, \ldots, h_n \rangle$ is well-formed. From the premise $e : \textbf{enum}_0$ we conclude that $\Gamma_r, v\colon u = \langle h_1, \ldots, h_n \rangle \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-ALLOC-UNION.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \textbf{union}(tv[h_1, \ldots, h_n], j) : t\ \textbf{in}\ e) \to (\Gamma_r, v\colon t = tv_j(h_1, \ldots, h_n) \mid \mathcal{C}_r \mid e)$$

The corresponding type judgment is TY-ALLOC-UNION.

$$\frac{\widehat{\Gamma_r} \vdash tv_j(a_1, \ldots, a_k) : \textbf{union}(tv[t_1, \ldots, t_m], \{j\})}{\widehat{\Gamma_r} \vdash \textbf{union}(tv[a_1, \ldots, a_k], j) : \textbf{alloc}\,\textbf{union}(tv[t_1, \ldots, t_m], \{j\})}$$

That is, $t = \textbf{union}(tv[t_1, \ldots, t_m], \{j\})$. From the premise of the typing rule, we can infer that $\Gamma_r, v\colon t = tv_i(h_1, \ldots, h_n)$ is a well-formed context, and from the premise $e : \textbf{enum}_0$ we conclude that $\Gamma_r, v\colon t = tv_i(h_1, \ldots, h_n) \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-ALLOC-MALLOC.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \textbf{malloc}(i_{size}) : t\ \textbf{in}\ e) \to (\Gamma_r, v\colon t = \langle \textsc{c} \rangle \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\, i_{size} \geq 0\end{array}$$

The corresponding type judgment is TY-ALLOC-MALLOC.

$$\frac{\widehat{\Gamma_r} \vdash h : \textbf{offset}}{\widehat{\Gamma_r} \vdash \textbf{malloc}(h) : \textbf{alloc}\,\textbf{data}}$$

That is $t = \textbf{data}$, and we can infer that the context $\Gamma_r, v\colon t = \langle \textsc{c} \rangle$ is well-formed. From the premise $e : \textbf{enum}_0$ we conclude that $\Gamma_r, v\colon t = \langle \textsc{c} \rangle \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-ALLOC-FRAME.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v = \textbf{frame}(tv[t_1, \ldots, t_m])\ \textbf{in}\ e) \to \\ \quad (\Gamma_r, v\colon \textbf{frame}(tv[t_1, \ldots, t_m]) = \langle \textsc{c} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle \mid \mathcal{C}_r \mid e)\end{array}$$

The corresponding type judgment is TY-ALLOC-FRAME.

$$\frac{\widehat{\Gamma_r} \vdash \textbf{frame}(tv[u_1, \ldots, u_m]) : \omega}{\widehat{\Gamma_r} \vdash \textbf{frame}(tv[u_1, \ldots, u_m]) : \textbf{alloc}\,\textbf{frame}(tv[u_1, \ldots, u_m])}$$

The premise of the typing rule requires that $tv[t_1, \ldots, t_m] : \omega_{frame}$. From the TY-STOREFRAME rule, we can infer that the context $\Gamma_r, v\colon \textbf{frame}(tv[t_1, \ldots, t_m]) = \langle \textsc{c} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle$ is well-formed.

$$\frac{\widehat{\Gamma_r} \vdash tv[t_1, \ldots, t_m] : \omega_{frame}}{\widehat{\Gamma_r} \vdash \langle \textsc{c} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle : \textbf{frame}(tv[t_1, \ldots, t_m])}$$

From the premise $e : \textbf{enum}_0$, we conclude that $\Gamma_r, v\colon \textbf{frame}(tv[\lceil t_i \rceil_1^m]) = \langle \textsc{c} : \textbf{frame}(tv[\lceil t_i \rceil_1^m]) \rangle \vdash e : \textbf{enum}_0$.

$\triangle$

LetSubscript There are several subscripting cases.

- Suppose the reduction uses the rule RED-LETSUB-TUPLE.

$$\begin{aligned}&((\Gamma'_r, v_2 : u' = \langle h_0, \ldots, h_{n-1}\rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow\\ &\quad (\Gamma_r, v_1 : u = h_j \mid \mathcal{C}_r \mid e)\end{aligned}$$

The proof of typing must end in the rule TY-LETSUB-TUPLE:

$$\frac{\widehat{\Gamma_r} \vdash v_2 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1}\rangle_c \quad \widehat{\Gamma_r}, v_1 : u \vdash e : \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e : \textbf{enum}_0}$$

From the premise $\Gamma_r \vdash v_2 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1}\rangle_c$, we can infer that u' is the tuple type $\langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1}\rangle_c$. In addition, the WELL-FORMED CONTEXT VALUES Lemma 6.1.4 implies that $\Gamma_r \vdash h_j : u$. Finally, from the premise $\Gamma_r, v_1 : u \vdash e : \textbf{enum}_0$ and the THIN-VAR-DEF rule, we can infer that $\Gamma_r, v_1 : u = h_j \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-LETSUB-ARRAY.

$$\begin{aligned}&((\Gamma'_r, v_2 : u' = \langle h_0, \ldots, h_{n-1}\rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow\\ &\quad (\Gamma_r, v_1 : u = h_j \mid \mathcal{C}_r \mid e)\\ &\quad \textbf{when}\, j \in \{0 \ldots n-1\}\end{aligned}$$

The corresponding type rule is TY-LETSUB-ARRAY.

$$\frac{\widehat{\Gamma_r} \vdash v_2 : u \textbf{ array} \quad \widehat{\Gamma_r} \vdash j : \textbf{offset} \quad \widehat{\Gamma_r}, v : u \vdash e : \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{let}\, v : u = v_2[j] \textbf{ in } e : \textbf{enum}_0}$$

From the premise $\widehat{\Gamma_r} \vdash v_2 : u \textbf{ array}$, we can infer that $u' = u \textbf{ array}$. In addition, the WELL-FORMED CONTEXT VALUES Lemma 6.1.4 implies that $\Gamma_r \vdash h_j : u$. Finally, from the premise $\Gamma_r, v_1 : u \vdash e : \textbf{enum}_0$ and the THIN-VAR-DEF rule, we can infer that $\Gamma_r, v_1 : u = h_j \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-LETSUB-UNION.

$$\begin{aligned}&((\Gamma'_r, v_2 : u' = tv_j(h_0, \ldots, h_{n-1})) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[k] \textbf{ in } e) \rightarrow\\ &\quad (\Gamma_r, v_1 : u = h_k \mid \mathcal{C}_r \mid e)\end{aligned}$$

The corresponding type rule is TY-LETSUB-UNION.

$$\frac{\begin{array}{l}\widehat{\Gamma_r} \vdash v_2 : \textbf{union}(tv[t'_1, \ldots, t'_m], \{j\})\\ \widehat{\Gamma_r} \vdash tv[t'_1, \ldots, t'_m] = \left(\lceil \vec{u}_i \rceil_0^{j-1} + \left\langle \lceil u'_i \rceil_0^{k-1}, u, \lceil u'_i \rceil_{k+1}^{l-1} \right\rangle + \lceil \vec{u}_i \rceil_{j+1}^{n-1}\right) : \omega_{union[n]}\\ \widehat{\Gamma_r}, v : u \vdash e : \textbf{enum}_0\end{array}}{\widehat{\Gamma_r} \vdash \textbf{let}\, v : u = v_2[k] \textbf{ in } e : \textbf{enum}_0}$$

From the premise $\widehat{\Gamma_r} \vdash v_2 : \textbf{union}(tv[t'_1, \ldots, t'_m], \{j\})$ we infer that $u' = \textbf{union}(tv[t'_1, \ldots, t'_m], \{j\})$. In addition, the WELL-FORMED CONTEXT VALUES Lemma 6.1.4 implies that $\Gamma_r \vdash h_k : u$. Finally, from the premise $\Gamma_r, v_1 : u \vdash e : \textbf{enum}_0$ and the THIN-VAR-DEF rule, we can infer that $\Gamma_r, v_1 : u = h_j \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-LETSUB-RAWDATA.

$$\begin{array}{l}((\Gamma'_r, v_2 : u' = \langle \mathrm{C} \rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow \\ \quad (\Gamma_r, v_1 : u = h \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\, \textbf{runtime}(\Gamma_r \mid \langle \mathrm{C} \rangle\, [j] : u) = h\end{array}$$

  By assumption, the runtime function $\textbf{runtime}(\Gamma_r \mid \langle \mathrm{C} \rangle\, [j] : u)$ establishes a derivation $\Gamma_r \vdash h : u$. From the THIN-VAR-DEF rule, we can infer that $\Gamma_r, v_1 : u = h \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-LETSUB-FRAME.

$$\begin{array}{l}((\Gamma'_r, v_2 : u' = \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow \\ \quad (\Gamma_r, v_1 : u = h \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\, \textbf{runtime}(\Gamma_r \mid \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle\, [j] : u) = h\end{array}$$

  By assumption, the runtime function $\textbf{runtime}(\Gamma_r \mid \langle \mathrm{C} : \textbf{frame}(tv[t_1, \ldots, t_m]) \rangle\, [j] : u)$ establishes a derivation $\Gamma_r \vdash h : u$. From the THIN-VAR-DEF rule, we can infer that $\Gamma_r, v_1 : u = h \vdash e : \textbf{enum}_0$.

△

SetSubscript There are several cases for subscript assignment on arrays.

- Suppose the reduction uses the rule RED-SETSUB-TUPLE.

$$\begin{array}{l}((\Gamma'_r, v : u' = \langle h_0, \ldots, h_{n-1} \rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid v[j] : u \leftarrow h;\ e) \rightarrow \\ \quad (\Gamma'_r, v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle \mid \mathcal{C}_r \mid e)\end{array}$$

  The proof of typing must end in the rule TY-SETSUB-TUPLE:

$$\frac{\widehat{\Gamma_r} \vdash v : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle_c \quad \widehat{\Gamma_r} \vdash h : u \quad \widehat{\Gamma_r} \vdash e : \textbf{enum}_0}{\widehat{\Gamma_r} \vdash v[j] : u \leftarrow h;\ e : \textbf{enum}_0}$$

  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 **array**. From the premise $h : u$, we know the context $\Gamma_r, v : u \textbf{ array} = \left\langle \lceil h_i \rceil_0^{i-1}, h, \lceil h_i \rceil_{i+1}^{n-1} \right\rangle$ is well-formed. From the premise $e : \textbf{enum}_0$ we conclude $\Gamma'_r, v : u = \left\langle \lceil h_i \rceil_0^{i-1}, h, \lceil h_i \rceil_{i+1}^{n-1} \right\rangle \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-SETSUB-ARRAY.

$$\begin{array}{l}((\Gamma'_r, v : u' = \langle h_0, \ldots, h_{n-1} \rangle) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid v[j] : u \leftarrow h;\ e) \rightarrow \\ \quad (\Gamma'_r, v : u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1} \rangle \mid \mathcal{C}_r \mid e) \\ \quad \textbf{when}\, j \in \{0 \ldots n-1\}\end{array}$$

  The proof of typing must end in the rule TY-SETSUB-ARRAY:

$$\frac{\widehat{\Gamma_r} \vdash v : u \textbf{ array} \quad \widehat{\Gamma_r} \vdash j : \textbf{offset} \quad \widehat{\Gamma_r} \vdash h : u \quad \widehat{\Gamma_r} \vdash e : \textbf{enum}_0}{\widehat{\Gamma_r} \vdash v[j] : u \leftarrow h;\ e : \textbf{enum}_0}$$

  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 $e : \textbf{enum}_0$ we conclude that $\Gamma'_r, v : u = \langle h_0, \ldots, h_{i-1}, h, h_{i+1}, \ldots, h_{n-1} \rangle \vdash e : \textbf{enum}_0$.

- Suppose the reduction uses the rule RED-SETSUB-UNION.

$$\begin{array}{l}(\Gamma_r, v\colon u' = tv_j(h_0, \ldots, h_{n-1}) \mid \mathcal{C}_r \mid v[k]\colon u \leftarrow h;\; e) \rightarrow \\ \qquad (\Gamma_r, v\colon u' = tv_j(h_0, \ldots, h_{k-1}, h, h_{k+1}, \ldots, h_{n-1}) \mid \mathcal{C}_r \mid e)\end{array}$$

The proof of typing must end in the rule TY-SETSUB-UNION:

$$\frac{\begin{array}{l}\widehat{\Gamma_r} \vdash v : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\}) \\ \widehat{\Gamma_r} \vdash tv[t'_1, \ldots, t'_m] = \left(\lceil \vec{u}_i \rceil_0^{j-1} + \left\langle \lceil u'_i \rceil_0^{k-1}, u, \lceil u'_i \rceil_{k+1}^{l-1} \right\rangle + \lceil \vec{u}_i \rceil_{j+1}^{n-1}\right) : \omega_{union[n]} \\ \widehat{\Gamma_r} \vdash h : u \\ \widehat{\Gamma_r} \vdash e : \mathbf{enum}_0\end{array}}{\widehat{\Gamma_r} \vdash v[k]\colon u \leftarrow h;\; e : \mathbf{enum}_0}$$

From the premise $\widehat{\Gamma_r} \vdash v : \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\})$ we infer that $u' = \mathbf{union}(tv[t'_1, \ldots, t'_m], \{j\})$. From the premise $h : u$, we know the context $\Gamma_r, v\colon u = tv_j(h_0, \ldots, h_{n-1})$ is well-formed. Finally, from the premise $e : \mathbf{enum}_0$ we conclude that $\Gamma'_r, v\colon u = tv_j(h_0, \ldots, h_{n-1}) \vdash e : \mathbf{enum}_0$.

- Suppose the reduction uses the rule RED-SETSUB-RAWDATA.

$$\begin{array}{l}((\Gamma'_r, v\colon u' = \langle \mathrm{C} \rangle) \;\mathbf{as}\; \Gamma_r \mid \mathcal{C}_r \mid v[j]\colon u \leftarrow h;\; e) \rightarrow \\ \qquad (\Gamma'_r, v\colon u' = \langle \mathrm{C}' \rangle \mid \mathcal{C}_r \mid e) \\ \qquad \mathbf{when\,runtime}(\Gamma_r \mid \langle \mathrm{C} \rangle\, [j] : u \leftarrow h) = \langle \mathrm{C}' \rangle\end{array}$$

The corresponding type rule is TY-SETSUB-RAWDATA.

$$\frac{\widehat{\Gamma_r} \vdash v : \mathbf{data} \quad \widehat{\Gamma_r} \vdash j : \mathbf{offset} \quad \widehat{\Gamma_r} \vdash h : u \quad \widehat{\Gamma_r} \vdash e : \mathbf{enum}_0}{\widehat{\Gamma_r} \vdash v[j]\colon u \leftarrow h;\; e : \mathbf{enum}_0}$$

We can infer that $u' = \mathbf{data}$. Since $\langle \mathrm{C}' \rangle : \mathbf{data}$, we conclude that $\Gamma'_r, v\colon \mathbf{data} = \langle \mathrm{C}' \rangle \vdash e : \mathbf{enum}_0$.

- Suppose the reduction uses the rule RED-SETSUB-FRAME.

$$\begin{array}{l}((\Gamma'_r, v\colon u' = \langle \mathrm{C} : \mathbf{frame}(tv[t_1, \ldots, t_m]) \rangle) \;\mathbf{as}\; \Gamma_r \mid \mathcal{C}_r \mid v[j]\colon u \leftarrow h;\; e) \rightarrow \\ \qquad (\Gamma'_r, v\colon u' = \langle \mathrm{C}' : \mathbf{frame}(tv[t_1, \ldots, t_m]) \rangle \mid \mathcal{C}_r \mid e) \\ \qquad \mathbf{when\,runtime}(\Gamma_r \mid \langle \mathrm{C} : \mathbf{frame}(tv[t_1, \ldots, t_m]) \rangle\, [j] : u \leftarrow h) \\ \qquad\qquad = \langle \mathrm{C}' : \mathbf{frame}(tv[t_1, \ldots, t_m]) \rangle\end{array}$$

The corresponding type rule is TY-SETSUB-FRAME.

$$\frac{\begin{array}{l}r_{\mathit{off}} \in \mathbb{Z}_{32}^{\mathbf{signed}} \\ \widehat{\Gamma_r} \vdash v : \mathbf{frame}(tv[t_1, \ldots, t_m]) \\ \widehat{\Gamma_r} \vdash tv[t_1, \ldots, t_m] = \left\{ l^f : \{ l^r : u; \cdots \}; \cdots \right\} : \omega_{frame} \\ \widehat{\Gamma_r} \vdash h : u \\ \widehat{\Gamma_r} \vdash e : \mathbf{enum}_0\end{array}}{\widehat{\Gamma_r} \vdash v[\mathbf{label}(tv, l^f, l^r, r_{\mathit{off}})]\colon u \leftarrow h;\; e : \mathbf{enum}_0}$$

We can infer that $u' = \mathbf{frame}(tv[\lceil t_i \rceil_1^m])$. Since $\langle \mathrm{C}' : \mathbf{frame}(tv[\lceil t_i \rceil_1^m]) \rangle : \mathbf{frame}(tv[t_1, \ldots, t_m])$, we conclude that $\Gamma'_r, v\colon u' = \langle \mathrm{C}' : \mathbf{frame}(tv[t_1, \ldots, t_m]) \rangle \vdash e : \mathbf{enum}_0$.

$\triangle$

LetGlobal Suppose that $e_r = \textbf{let}\, v\colon u = \textbf{global}\, l \textbf{ in } e$. The operational rule is RED-LETGLOBAL.

$$((\Gamma'_r, l\colon u = h) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v\colon u = \textbf{global}\, l \textbf{ in } e) \to (\Gamma_r, v\colon u = h \mid \mathcal{C}_r \mid e)$$

The proof of typing must end in the rule TY-LETGLOBAL.

$$\frac{\widehat{\Gamma_r} \vdash l : u \quad \widehat{\Gamma_r}, v\colon u \vdash e : \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{let}\, v\colon u = \textbf{global}\, l \textbf{ in } e : \textbf{enum}_0}$$

From the premise $l : u$, we know that $u' = u$. From the premise $\Gamma_r, v\colon u \vdash e : \textbf{enum}_0$, we conclude that $\Gamma_r, v\colon u = h \vdash e : \textbf{enum}_0$. △

SetGlobal Suppose that $e_r = \textbf{global}\, l\colon u \leftarrow h;\ e$. The operational rule is RED-SETGLOBAL.

$$((\Gamma'_r, l\colon u = h') \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{global}\, l\colon u \leftarrow h;\ e) \to (\Gamma'_r, l\colon u = h \mid \mathcal{C}_r \mid e)$$

The proof of typing must end in the rule TY-SETGLOBAL.

$$\frac{\widehat{\Gamma_r} \vdash l : u \quad \widehat{\Gamma_r} \vdash a : u \quad \widehat{\Gamma_r} \vdash e : \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{global}\, l\colon u \leftarrow a;\ e : \textbf{enum}_0}$$

From the premise $l : u$, we know that $u' = u$. From the premise $\Gamma_r \vdash e : \textbf{enum}_0$, we conclude that $\Gamma'_r, l\colon u = h \vdash e : \textbf{enum}_0$. △

□

7.2 Progress

Theorem 7.2.1 PROGRESS *If* $(\Gamma_r \mid \mathcal{C}_r \mid e_r)$ *is a program, and* e_r *is not a value* h*, then there is a program* $(\Gamma_c \mid \mathcal{C}_c \mid e_c)$ *such that* $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to (\Gamma_c \mid \mathcal{C}_c \mid e_c)$*, or* $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to \textbf{error}$.

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 $v\colon t$ rather than value *definitions* of the form $v\colon t = b$. We use the PROOF INDUCTION Lemma 6.2.1 throughout this proof to infer use of specific type rules in a partially thinned context $\widehat{\Gamma_r}$ where some number of thinning rules have been applied.

Variables Suppose $e_r = E[v]$ and $\Gamma_r = (\Gamma'_r, v\colon t' = h)$. The rule RED-ATOM-VAR applies.

$$(\Gamma_r, v\colon t = h \mid \mathcal{C}_r \mid E[v]) \to (\Gamma_r, v\colon t = h \mid \mathcal{C}_r \mid E[h])$$

In the remainder of the proof, we will consider only the cases for $E[v]$ where $\Gamma_r \neq (\Gamma'_r, v\colon t' = h)$. △

Unary operations Suppose $e_r = E[\mathit{unop}\, h]$. By the EXPRESSION TYPING Lemma 6.4.4, the atom $\mathit{unop}\, h$ must be well-typed for some type t_a. The only rule that applies is TY-ATOMUNOP.

$$\frac{\widehat{\Gamma_r} \vdash a : \mathbf{arg}(\mathit{unop}) \quad \widehat{\Gamma_r} \vdash \mathbf{res}(\mathit{unop}) : \Omega}{\widehat{\Gamma_r} \vdash \mathit{unop}\, a : \mathbf{res}(\mathit{unop})}$$

According to the unary operator syntax table in Figure 2.5, there are two general cases for $\mathbf{arg}(\mathit{unop})$: it is either a scalar type, or the unary operator is a type coercion.

- In the type coercion case, the unary reduction RED-ATOM-UNOP always applies.
- To illustrate the scalar case, suppose $\mathbf{arg}(\mathit{unop}) = \mathbb{Z}_{31}$. 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 RED-ATOM-UNOP applies. Otherwise, if h is a variable v, then it must be defined in the context Γ_r with a definition $v \colon \mathbb{Z}_{31} = b$. 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 RED-ATOM-VAR reduction applies.

$\triangle$

Binary operations Binary operations are similar to the unary case. Suppose $e_r = E[h_1\ \mathit{binop}\ h_2]$. By the EXPRESSION TYPING Lemma 6.4.4, the atom $h_1\ \mathit{binop}\ h_2$ must be well-typed for some type t_a. The only rule that applies is TY-ATOMBINOP.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : \mathbf{arg}_1(\mathit{binop}) \quad \widehat{\Gamma_r} \vdash a_2 : \mathbf{arg}_2(\mathit{binop}) \quad \widehat{\Gamma_r} \vdash \mathbf{res}(\mathit{binop}) : \Omega}{\widehat{\Gamma_r} \vdash a_1\ \mathit{binop}\ a_2 : \mathbf{res}(\mathit{binop})}$$

According to the binary operator syntax table in Figure 2.7, there are two general cases for $\mathbf{arg}_j(\mathit{binop})$: it is either a scalar type, or the binary operator is a comparison. In the comparison case, the binary reduction RED-ATOM-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 RED-ATOM-BINOP or RED-ATOM-VAR applies. $\triangle$

Type application Suppose $e_r = E[\mathbf{ty\_apply}[u](h_f, u_1, \ldots, u_n)]$. By EXPRESSION TYPING Lemma 6.4.4, the atom $\mathbf{ty\_apply}[u](h_f, u_1, \ldots, u_n)$ must be well-typed for some type t_a. The only rule that applies is the TY-ATOMTYAPPLY rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil u_i : \omega \rceil_1^m \quad \widehat{\Gamma_r} \vdash h_f : \forall \alpha_1, \ldots, \alpha_m.t \quad \widehat{\Gamma_r} \vdash u = t[\lceil u_i/\alpha_i \rceil_1^m] : \omega}{\widehat{\Gamma_r} \vdash \mathbf{ty\_apply}[u](h_f, u_1, \ldots, u_m) : t[\lceil u_i/\alpha_i \rceil_1^m]}$$

The only rule that can be used to prove the premise $\widehat{\Gamma_r} \vdash h_f : \forall \alpha_1, \ldots, \alpha_n.t$ is the TY-ATOMVAR rule.

$$\frac{\widehat{\Gamma_r}, v \colon \forall \alpha_1, \ldots, \alpha_n.t \vdash \diamond}{\widehat{\Gamma_r}, v \colon \forall \alpha_1, \ldots, \alpha_n.t \vdash v : \forall \alpha_1, \ldots, \alpha_n.t}$$

That is, the value h_f must be a variable v, and context Γ_r must include the declaration $v \colon \forall \alpha_1, \ldots, \alpha_n.t$. Since the context is fully-defined and well-formed, it contains a definition $v \colon \forall \alpha_1, \ldots, \alpha_n.t_a = b$, which is well-formed using the judgment THIN-VAR-DEF.

$$\frac{\widehat{\Gamma_r}, v\colon \forall\alpha_1,\ldots,\alpha_n.t \vdash b : \forall\alpha_1,\ldots,\alpha_n.t \quad \widehat{\Gamma_r}, v\colon \forall\alpha_1,\ldots,\alpha_n.t \vdash \diamond}{\widehat{\Gamma_r}, v\colon \forall\alpha_1,\ldots,\alpha_n.t = b \vdash \diamond}$$

The only possible value b for v is specified by the TY-STOREFUN-POLY rule.

$$\frac{\widehat{\Gamma_r}, \lceil\alpha_i : \omega\rceil_1^m \vdash \lambda v_1,\ldots,v_n.e : (u_1,\ldots,u_n) \to t}{\widehat{\Gamma_r} \vdash \Lambda\alpha_1,\ldots,\alpha_m.\lambda v_1,\ldots,v_n.e : \forall\alpha_1,\ldots,\alpha_m.(u_1,\ldots,u_n) \to t}$$

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 Γ_r has the form

$$\Gamma_r = \Gamma'_r, f\colon \forall\alpha_1,\ldots,\alpha_m.(u_1,\ldots,u_n) \to t = \Lambda\alpha_1,\ldots,\alpha_m.\lambda v_1,\ldots,v_n.e.$$

The operational rule RED-ATOM-APPLY applies.

$$\begin{array}{l}((\Gamma'_r, f\colon u' = \Lambda\alpha_1,\ldots,\alpha_m.b) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid E[\textbf{ty\_apply}[u](f,u_1,\ldots,u_m)]) \to \\ \quad (\Gamma_r, \lceil\alpha_i : \omega = u_i\rceil_1^m, g\colon u = b \mid \mathcal{C}_r \mid E[g])\end{array}$$

$\triangle$

Type abstraction Suppose $e_r = E[\textbf{ty\_pack}[u](v_1,u_1,\ldots,u_{n-1})]$. By EXPRESSION TYPING Lemma 6.4.4, the atom $\textbf{ty\_pack}[u](v_1,u_1,\ldots,u_{n-1})$ must be well-typed for some type t_a. The only rule that applies is the TY-ATOMTYPACK rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil u_i : \omega\rceil_1^m \quad \widehat{\Gamma_r} \vdash v_1 : t[\lceil u_i/\alpha_i\rceil_1^m] \quad \widehat{\Gamma_r} \vdash u = \exists\alpha_1,\ldots,\alpha_m.t : \omega}{\widehat{\Gamma_r} \vdash \textbf{ty\_pack}[u](v_1,u_1,\ldots,u_m) : \exists\alpha_1,\ldots,\alpha_m.t}$$

The only rule that can be used to prove the premise $\widehat{\Gamma_r} \vdash v_1 : t[\lceil u_i/\alpha_i\rceil_1^m]$ is the TY-ATOMVAR rule.

$$\frac{\widehat{\Gamma_r}, v\colon t[\lceil u_i/\alpha_i\rceil_1^m] \vdash \diamond}{\widehat{\Gamma_r}, v\colon t[\lceil u_i/\alpha_i\rceil_1^m] \vdash v : t[\lceil u_i/\alpha_i\rceil_1^m]}$$

That is, atom a must be a variable, and context Γ_r must include the declaration $v\colon t[\lceil u_i/\alpha_i\rceil_1^m]$. Since the context is fully-defined and well-formed, it contains a definition $v\colon t[\lceil u_i/\alpha_i\rceil_1^n] = b$. The operational rule RED-ATOM-PACK applies.

$$\begin{array}{c}((\Gamma'_r, v_1 : u' = b) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid E[\textbf{ty\_pack}[u](v_1,u_0,\ldots,u_{m-1})]) \to \\ (\Gamma_r, v_2\colon u = \textbf{ty\_pack}[u](v_1,u_0,\ldots,u_{m-1}) \mid \mathcal{C}_r \mid E[v_2])\end{array}$$

$\triangle$

Abstraction unpacking Suppose $e_r = E[\textbf{ty\_unpack}(v_1)]$. By the EXPRESSION TYPING Lemma 6.4.4, the atom $\textbf{ty\_unpack}(v_1)$ must be well-typed for some type t_a. The only rule that can be applied is TY-ATOMTYUNPACK.

$$\frac{\widehat{\Gamma_r} \vdash v : \exists \alpha_0, \ldots, \alpha_{m-1}.t}{\widehat{\Gamma_r} \vdash \textbf{ty\_unpack}(v) : t[\lceil v.i/\alpha_i \rceil_0^{m-1}]}$$

The only rule that can be used to prove the premise $\widehat{\Gamma_r} \vdash v : \exists \alpha_1, \ldots, \alpha_m.t$ is the Ty-AtomVar rule.

$$\frac{\widehat{\Gamma_r}, v \colon \exists \alpha_1, \ldots, \alpha_m.t \vdash \diamond}{\widehat{\Gamma_r}, v \colon \exists \alpha_1, \ldots, \alpha_m.t \vdash v : \exists \alpha_1, \ldots, \alpha_m.t}$$

That is, the context Γ_r must include the declaration $v \colon \exists \alpha_1, \ldots, \alpha_n.t$. Since the context is fully-defined and well-formed, it contains a definition $v \colon \exists \alpha_1, \ldots, \alpha_n.t = b$, which is well-formed using the judgment Thin-Var-Def.

$$\frac{\widehat{\Gamma_r}, v \colon \exists \alpha_1, \ldots, \alpha_m.t \vdash b : \exists \alpha_1, \ldots, \alpha_m.t \quad \widehat{\Gamma_r}, v \colon \exists \alpha_1, \ldots, \alpha_m.t \vdash \diamond}{\widehat{\Gamma_r}, v \colon \exists \alpha_1, \ldots, \alpha_m.t = b \vdash \diamond}$$

The only possible value b for v is specified by the Ty-AtomTyPack rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil u_i : \omega \rceil_1^m \quad \widehat{\Gamma_r} \vdash v : t[\lceil u_i/\alpha_i \rceil_1^m] \quad \widehat{\Gamma_r} \vdash u = \exists \alpha_1, \ldots, \alpha_m.t : \omega}{\widehat{\Gamma_r} \vdash \textbf{ty\_pack}[u](v, u_1, \ldots, u_m) : \exists \alpha_1, \ldots, \alpha_m.t}$$

That is, b must be a packed expression $\textbf{ty\_pack}[u](v, u_1, \ldots, u_m)$. This means that the context Γ_r has the form $\Gamma_r', v \colon \exists \alpha_1, \ldots, \alpha_m.t = \textbf{ty\_pack}[u](v, u_1, \ldots, u_m)$. The operational rule Red-Atom-Unpack applies.

$$\begin{array}{l}\Big((\Gamma_r', v_1 : u = \textbf{ty\_pack}[\exists \lceil \alpha_i \rceil_0^{m-1}.t](v, \lceil u_i \rceil_0^{m-1}))\ \textbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid E[\textbf{ty\_unpack}(v_1)]\Big) \rightarrow \\ \quad \Big(\Gamma_r, \lceil \alpha_i \colon \omega = u_i \rceil_0^{m-1}, v_2 \colon t = v \mid \mathcal{C}_r \mid E[v_2]\Big)\end{array}$$

$\triangle$

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.

LetAtom Suppose $e_r = \textbf{let}\, v \colon t = h\ \textbf{in}\ e$. The reduction Red-LetAtom applies.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v \colon u = h\ \textbf{in}\ e) \rightarrow (\Gamma_r, v \colon u = h \mid \mathcal{C}_r \mid e)$$

$\triangle$

TailCall Suppose $e_r = h(h_1, \ldots, h_n)$. The type judgment must use the Ty-TailCall rule.

$$\frac{\widehat{\Gamma_r} \vdash h : (u_1, \ldots, u_n) \rightarrow \textbf{enum}_0 \quad \widehat{\Gamma_r} \vdash \lceil h_i : u_i \rceil_1^n}{\widehat{\Gamma_r} \vdash h(h_1, \ldots, h_n) : \textbf{enum}_0}$$

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.

$$\begin{array}{l}((\Gamma'_r, v\colon (u_1, \ldots, u_n) \to t = \lambda v_1, \ldots, v_n.e) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid v(h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r, \lceil v_i\colon u_i = h_i \rceil_1^n \mid \mathcal{C}_r \mid e)\end{array}$$

$\triangle$

Special Calls Suppose $e_r = \textbf{special}\, S$ is a special call. There are several cases.

- Suppose $e_r = \textbf{migrate}\ [j, h_p, h_o]\ h_{fun}(h_1, \ldots, h_n)$. The typing rule is Ty-SysMigrate.

$$\frac{\begin{array}{cc} & \widehat{\Gamma_r} \vdash h_{ptr} : \textbf{data} \\ j \in \mathbb{Z}_{31} & \widehat{\Gamma_r} \vdash h_{off} : \mathbb{Z}_{32}^{\textbf{signed}} \\ \widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n & \widehat{\Gamma_r} \vdash h_f : (t_1, \ldots, t_n) \to \textbf{enum}_0 \end{array}}{\widehat{\Gamma_r} \vdash \textbf{migrate}\ [j, h_{ptr}, h_{off}]\ h_f(h_1, \ldots, h_n) : \textbf{special}\,\textbf{enum}_0}$$

  By the Function Values Lemma 6.7.3, since $h_f : (t_1, \ldots, t_n) \to \textbf{enum}_0$, it must be defined in the context as a function $\lambda v_1, \ldots, v_n.e$. The Red-SysMigrate rule applies.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{special}\,\textbf{migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)) \to (\Gamma_r \mid \mathcal{C}_r \mid f(h_1, \ldots, h_n))$$

- Suppose $e_r = \textbf{atomic}\ h_f(h_{const}, h_1, \ldots, h_n)$. The corresponding typing rule is Ty-Atomic.

$$\frac{\widehat{\Gamma_r} \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash h_f : (\mathbb{Z}_{32}^{\textbf{signed}}, t_1, \ldots, t_n) \to \textbf{enum}_0}{\widehat{\Gamma_r} \vdash \textbf{atomic}\ h_f(h_{const}, h_1, \ldots, h_n) : \textbf{special}\,\textbf{enum}_0}$$

  By the Function Values Lemma 6.7.3, since $h_f : (h_{const}, h_1, \ldots, h_n) \to \textbf{enum}_0$, the value h_f must be a variable f. The Red-Atomic rule applies.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{special}\,\textbf{atomic}\ f(h_{const}, h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r \mid \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle\,; \mathcal{C}_r \mid f(h_{const}, h_1, \ldots, h_n))\end{array}$$

- Suppose $e_r = \textbf{rollback}\ [h_{level}, h_{const}]$. The correspoonding typing rule is Ty-AtomicRollback.

$$\frac{\widehat{\Gamma_r} \vdash h_{level} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \widehat{\Gamma_r} \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}}}{\widehat{\Gamma_r} \vdash \textbf{rollback}\ [h_{level}, h_{const}] : \textbf{special}\,\textbf{enum}_0}$$

  By the Numeric Values Lemma 6.7.2, the value h_{level} must be a variable or a constant. In the former case, the Red-Atom-Var rule applies. Otherwise, let $i = h_{level}$, and suppose the checkpoint context $\mathcal{C}_r$ contains m checkpoints $C_m, \ldots, C_1$.
  If $1 \le i \le m$, then the Red-Atomic-Rollback-1 rule applies.

$$\begin{array}{l}(\Gamma'_r \mid C_m; \ldots; C_i = \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle\,; \ldots; C_1 \mid \textbf{special}\,\textbf{rollback}\ [i, j]) \to \\ \quad (\Gamma_r \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, h_1, \ldots, h_n)) \\ \quad \textbf{when}\, i \in \{1 \ldots m\}\end{array}$$

  If $i = 0 \wedge m > 0$, then the Red-Atomic-Rollback-2 rule applies.

$$\begin{array}{l}(\Gamma'_r \mid C_m = \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle\,; \ldots; C_1 \mid \textbf{special}\,\textbf{rollback}\ [0, j]) \to \\ \quad (\Gamma_r \mid C_m; \ldots; C_1 \mid f(j, h_1, \ldots, h_n))\end{array}$$

If $i < 0 \vee i > m$, then the RED-ATOMIC-ROLLBACK-ERROR rule applies.

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots; C_1 \mid \mathbf{special\,rollback}\ [i, j]) \rightarrow \mathbf{error} \\ \quad \mathbf{when}\, i \notin \{0 \ldots m\} \vee m = 0\end{array}$$

- Suppose $e_r = \mathbf{commit}\ [h_{level}]\ h_f(h_1, \ldots, h_n)$. The typing rule is TY-ATOMICCOMMIT.

$$\frac{\widehat{\Gamma_r} \vdash h_{level} : \mathbb{Z}_{32}^{\mathbf{signed}} \quad \widehat{\Gamma_r} \vdash \lceil h_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash h_f : (t_1, \ldots, t_n) \rightarrow \mathbf{enum}_0}{\widehat{\Gamma_r} \vdash \mathbf{commit}\ [h_{level}]\ h_f(h_1, \ldots, h_n) : \mathbf{special\,enum}_0}$$

By the NUMERIC VALUES Lemma 6.7.2, the value h_{level} must be a variable or a constant. In the former case, the RED-ATOM-VAR rule applies. Otherwise, let $i = h_{level}$, and suppose the checkpoint context $\mathcal{C}_r$ contains m checkpoints $C_m, \ldots, C_1$.
If $1 \leq i \leq m$, then the RED-ATOMIC-COMMIT-1 rule applies.

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots; C_1 \mid \mathbf{special\,commit}\ [i]\ f(h_1, \ldots, h_n)) \rightarrow \\ \quad (\Gamma_r \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) \\ \quad \mathbf{when}\, i \in \{1 \ldots m\}\end{array}$$

If $i = 0 \wedge m > 0$, then the RED-ATOMIC-COMMIT-2 rule applies.

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots; C_1 \mid \mathbf{special\,commit}\ [0]\ f(h_1, \ldots, h_n)) \rightarrow \\ \quad (\Gamma_r \mid C_{m-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n))\end{array}$$

If $i < 0 \vee i > m$, then the RED-ATOMIC-COMMIT-ERROR rule applies.

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots; C_1 \mid \mathbf{special\,commit}\ [i]\ f(h_1, \ldots, h_n)) \rightarrow \mathbf{error} \\ \quad \mathbf{when}\, i \notin \{0 \ldots m\} \vee m = 0\end{array}$$

△

Match Suppose $e_r = \mathbf{match}\, h\ \mathbf{with}\ \lceil s_j \mapsto e_j \rceil_{j=1}^n$. The type judgment must use one of the match rules. We illustrate with the rule TY-MATCH-INT.

$$\frac{\lceil s_i \in set_I \rceil_1^n \quad \bigcup_{i=1}^n s_i = \mathbb{Z}_{31} \quad \widehat{\Gamma_r} \vdash a : \mathbb{Z}_{31} \quad \widehat{\Gamma_r} \vdash \lceil e_i : t \rceil_1^n}{\widehat{\Gamma_r} \vdash \mathbf{match}\, a\ \mathbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n : t}$$

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 $\bigcup_{j=1}^n s_j = \mathbb{Z}_{31}$. Since h must belong to *one* of the sets s_k, the operation rule RED-MATCH-INT applies.

$$\begin{array}{l}\left(\Gamma_r \mid \mathcal{C}_r \mid \mathbf{match}\, i\ \mathbf{with}\ \lceil s_j \mapsto e_j \rceil_{j=1}^n\right) \rightarrow (\Gamma_r \mid \mathcal{C}_r \mid e_k) \\ \quad \mathbf{when}\ (i \in s_k) \wedge \forall j \in \{1, \ldots, k-1\}.(i \notin s_j)\end{array}$$

△

Allocation Suppose $e_r = \mathbf{let}\, v = \mathit{alloc}\ \mathbf{in}\ e$. One of the allocation reductions RED-ALLOC-TUPLE, RED-ALLOC-UNION, or RED-ALLOC-ARRAY always applies. △

Subscripting Suppose $e_r = \mathbf{let}\, v\colon t = h_1[h_2]\ \mathbf{in}\ e$. The type judgment requires application of one of the subscripting type rules. We illustrate with TY-LETSUB-TUPLE.

$$\frac{\widehat{\Gamma_r} \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1}\rangle_c \quad \widehat{\Gamma_r}, v_1 : u \vdash e : t}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v_1 : u = a_1[\mathbf{offset}_c^{\mathbf{const}}(j)]\ \mathbf{in}\ e : t}$$

In this case, h_1 must be a variable v_1 with type $\langle u_0, \ldots, u_{i-1}, t_1, u_{i+1}, \ldots, u_{n-1}\rangle_c$, and h_2 must be a constant $\mathbf{offset}_c^{\mathbf{const}}(i)$.

Since v_1 has a tuple type, its value is determined by the TY-STORETUPLE rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil a_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash \langle t_1, \ldots, t_n\rangle_{tuple\_class} : \Omega}{\widehat{\Gamma_r} \vdash \langle a_1, \ldots, a_n\rangle : \langle t_1, \ldots, t_n\rangle_{tuple\_class}}$$

That is v_1 must be a tuple $v_1\colon \langle u_0, \ldots, u_{i-1}, t_1, u_{i+1}, \ldots, u_{n-1}\rangle_c = \langle a_1, \ldots, a_n\rangle$.

The reduction RED-LETSUB-TUPLE applies.

$$\begin{aligned}&((\Gamma'_r, v_2 \colon u' = \langle h_0, \ldots, h_{n-1}\rangle)\ \mathbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid \mathbf{let}\, v_1 : u = v_2[j]\ \mathbf{in}\ e) \rightarrow \\ &\quad (\Gamma_r, v_1 : u = h_j \mid \mathcal{C}_r \mid e)\end{aligned}$$

$\triangle$

Assignment Suppose $e_r = h_1[h_2]\colon t \leftarrow h_3;\ e$. The type judgment requires application of one of the subscript assignment type rules. We illustrate with TY-SETSUB-TUPLE.

$$\frac{\widehat{\Gamma_r} \vdash h_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1}\rangle_c \quad \widehat{\Gamma_r} \vdash a_3 : u \quad \widehat{\Gamma_r} \vdash e : t}{\widehat{\Gamma_r} \vdash h_1[\mathbf{offset}_c^{\mathbf{const}}(j)]\colon u \leftarrow a_3;\ e : t}$$

In this case, h_1 must be a variable v_1 with type $\langle u_0, \ldots, u_{i-1}, t_1, u_{i+1}, \ldots, u_{n-1}\rangle_c$, and h_2 must be a constant $\mathbf{offset}_c^{\mathbf{const}}(i)$.

Since v_1 has a tuple type, its value is determined by the TY-STORETUPLE rule.

$$\frac{\widehat{\Gamma_r} \vdash \lceil a_i : t_i \rceil_1^n \quad \widehat{\Gamma_r} \vdash \langle t_1, \ldots, t_n\rangle_{tuple\_class} : \Omega}{\widehat{\Gamma_r} \vdash \langle a_1, \ldots, a_n\rangle : \langle t_1, \ldots, t_n\rangle_{tuple\_class}}$$

That is v_1 must be a tuple $v_1\colon \langle u_0, \ldots, u_{i-1}, t_1, u_{i+1}, \ldots, u_{n-1}\rangle_c = \langle a_1, \ldots, a_n\rangle$.

The reduction RED-SETSUB-TUPLE applies.

$$\begin{aligned}&((\Gamma'_r, v\colon u' = \langle h_0, \ldots, h_{n-1}\rangle)\ \mathbf{as}\ \Gamma_r \mid \mathcal{C}_r \mid v[j]\colon u \leftarrow h;\ e) \rightarrow \\ &\quad (\Gamma'_r, v\colon u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1}\rangle \mid \mathcal{C}_r \mid e)\end{aligned}$$

$\triangle$

LetGlobal Suppose $e_r = \mathbf{let}\, v\colon t = \mathbf{global}\, l\ \mathbf{in}\ e$. The corresponding typing rule is TY-LETGLOBAL.

$$\frac{\widehat{\Gamma_r} \vdash l : u \quad \widehat{\Gamma_r}, v\colon u \vdash e : t}{\widehat{\Gamma_r} \vdash \mathbf{let}\, v\colon u = \mathbf{global}\, l\ \mathbf{in}\ e : t}$$

By the VARIABLE DECLARATIONS Lemma 6.7.1, the label l must be defined as part of the context. The RED-LETGLOBAL rule applies.

$$((\Gamma'_r, l\colon u = h) \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{let}\, v\colon u = \textbf{global}\, l \textbf{ in } e) \to (\Gamma_r, v\colon u = h \mid \mathcal{C}_r \mid e)$$

$\triangle$

SetGlobal Suppose $e_r = \textbf{global}\, l\colon t \leftarrow h;\ e$. The corresponding typing rule is TY-SETGLOBAL.

$$\frac{\widehat{\Gamma_r} \vdash l : u \quad \widehat{\Gamma_r} \vdash a : u \quad \widehat{\Gamma_r} \vdash e : t}{\widehat{\Gamma_r} \vdash \textbf{global}\, l\colon u \leftarrow a;\ e : t}$$

By the VARIABLE DECLARATIONS Lemma 6.7.1, the label l must be defined as part of the context. The RED-SETGLOBAL rule applies.

$$((\Gamma'_r, l\colon u = h') \textbf{ as } \Gamma_r \mid \mathcal{C}_r \mid \textbf{global}\, l\colon u \leftarrow h;\ e) \to (\Gamma'_r, l\colon u = h \mid \mathcal{C}_r \mid e)$$

$\triangle$

$\square$

Chapter 8

Runtime Implementation

The FIR is machine-independent, and the Mojave compiler architecture is designed to support multiple back-ends, 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 **data** and $\mathbf{frame}(tv[t_1, \ldots, t_n])$. Safe data is type-safe ML data, and corresponds to the $\langle t_1, \ldots, t_n \rangle_{tuple\_class}$, t **array**, and $\mathbf{union}(tv[t_1, \ldots, t_n], set_I)$ 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 *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 `-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.

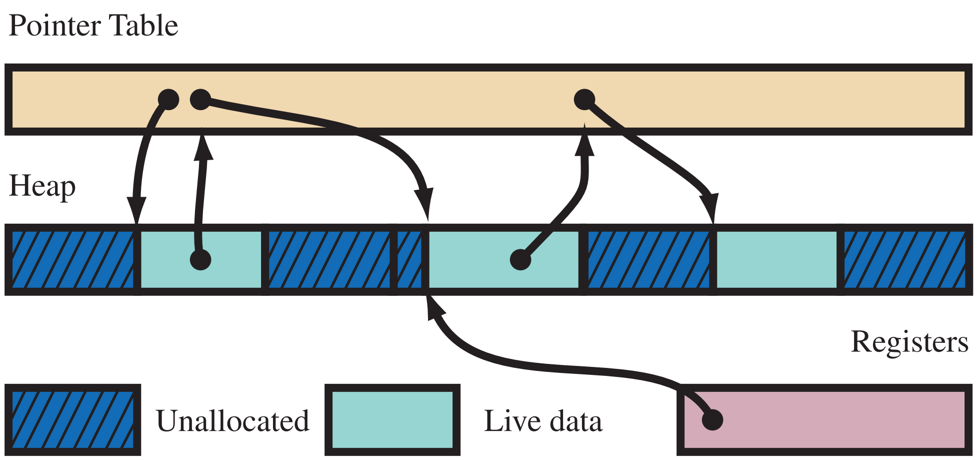

Figure 8.1: Pointer table representation

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 *funtable* 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)[5]. 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[6]. 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[7]. 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 $\mathbf{runtime}(\Gamma \mid \langle \mathrm{C} \rangle\,[i] : t)$ and store $\mathbf{runtime}(\Gamma \mid \langle \mathrm{C} \rangle\,[i] : t \leftarrow h)$ 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 $\langle \mathrm{C} \rangle$; 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 $funtable[h]$.
   - 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 $\mathbf{runtime}(\Gamma \mid \langle \mathrm{C} \rangle\,[i] : t \leftarrow h)$, 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 aggr`, `pointer block`, `poly`, and `function` $\diamond$.

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** $[i, a_{ptr}, a_{off}]\ f(a_1, \ldots, a_n)$. 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_{ptr}, a_{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, `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 `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[9]. 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*[10]. 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] 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 **commit** $[level]$ $f(a_1, \ldots, a_n)$. This will commit data for *level* by folding all changes from that level into its previous level. *level* must be in the interval $\{0 \ldots N\}$, otherwise a runtime exception will occur. If $level = 0$, then the most recent level N is committed.

The primitive for *rollback* is **rollback** $[level, c]$. This will revert all changes made by in level *level* and all later levels. *level* must be in the interval $\{0 \ldots N\}$, otherwise a runtime exception will occur. If $level = 0$, then the most recent level N is reverted.

Rollback resumes execution at the point where *level* was entered. No function or argument list is specified; the function that was associated with level *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 *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 $base[i]$, and a limit pointer $base[i+1]$, or, in the case of the youngest generation, *limit*.
- The upper bound of atomic level i is delimited by the $level[i]$ pointer.

  Invariant: (ATOMIC INVARIANT) all heap data for atomic level i is between $base[1]$ and $level[i]$, 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 *level* was entered.

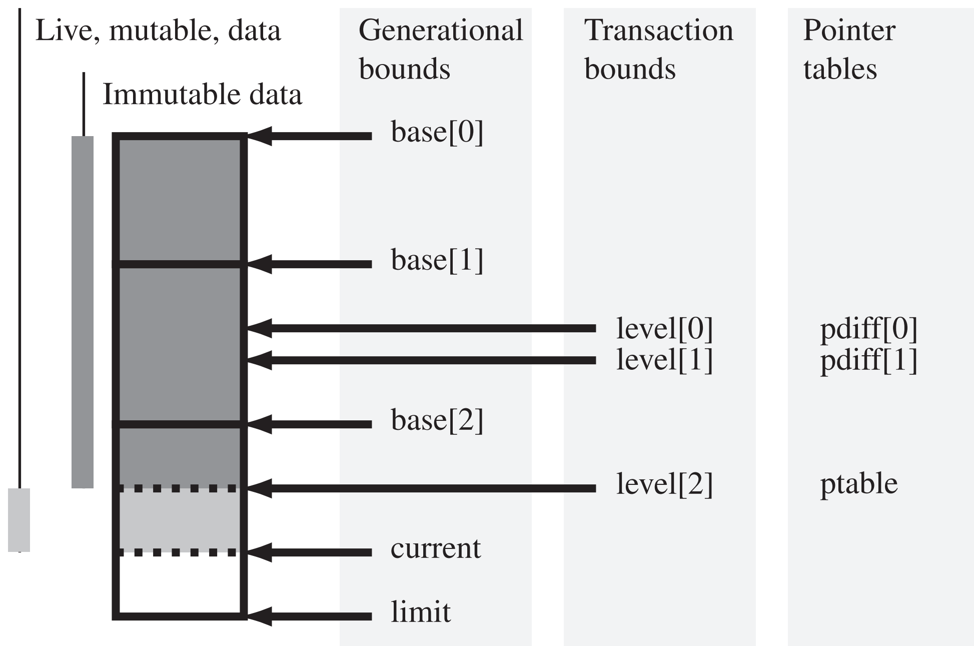

Figure 8.2: Heap data with multiple atomic levels

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 $level[i]$ and the difference table $pdiff[i]$ 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 $pdiff[i]$ table, which is deleted along with the $level[i]$ 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 $level = 0$. 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

[1] A.W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. 1

[2] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 365–377, 2000. 1.1

[3] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, pages 198–229, 2001. Special Issue on Coordination. 1.1

[4] G. Di Marzo Serugendo, M. Muhugusa, and C. Tschudin. A survey of theories for mobile agents. *World Wide Web Journal, special issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents*, 1998. 1.1

[5] J. Engel. *Programming for the Java Virtual Machine*. Addison Wesley, 1999. 1.1

[6] Lal George. ML RISC: Customizable and reusable code generators. www.cs.bell-labs.com/~george, 1996. 1.1

[7] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et sonapplication a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–69. Springer-Verlag, NY, 1971. 1

[8] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *POPL*, 2001. 1.1

[9] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, November 1994. Short Communication. 1.1

[10] J.G.Morrisett, T.Jim, D.Grossman, M.Hicks, J.Cheney, and Y.Wang. Cyclone: A safe dialect of C. In *Usenix Annual Technical Conference*, 2002. 1.1

[11] Simon Peyton Jones, D. Oliva, and T. Nordin. C--: a portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*, 1998. 1.1

[12] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *PPDP*, 1999. Invited talk. 1.1

[13] S. Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation Research Institute, 1993. 1.1

[14] Erik Meijer and John Gough. A technical overview of the common language infrastructure. http://research.microsoft.com/~emeijer. 1.1

[15] Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. (Workshop on Compiler Support for Systems Software, Tucson, Arizona.), February 1996. 1

[16] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, 2002. 1.1

[17] Richard M. Stallman. Using and porting GNU CC (version 2.0). Technical report, Free Software Foundation, 1992. 1.1

[18] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996. 1

[19] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, 1998. 10

[20] Giovanni Vigna, editor. *Mobile Agents and Security*. Springer, 1999. LNCS 1419. 1.1