Thesis Proposal:
Certifying compilation for Standard ML in a type analysis framework.

Leaf Petersen

September 2001

Abstract

Certified code is native machine code that is annotated with an automatically checkable certificate attesting to the conformance of the program to a specified safety policy. Certified code frameworks have been built based on first-order logic (PCC) and on types (TAL). Compilers generating certified code have been built for safe subsets of C and for Java(tm).

Type-preserving compilers such as the TILT/ML compiler implement compilation as transformations on *typed* internal languages. Types are used by the compiler for internal verification, and for optimization purposes. Type analysis can be used to implement optimizations such as non-uniform data representation and tag-free garbage collection. However, none of the existing type-preserving compilers for full-scale languages maintain type information all the way to the machine-code level, and hence are not yet able to generate certified code.

I propose to extend the TILT/ML compiler to generate certified code in the form of typed assembly language without compromising the existing optimizations of the compiler. *The goal of this work is to show that a compiler can use types to generate certified binaries for a full modern language even in the presence of agressive type-analysis based optimization.*

1 Introduction

It is an unfortunate fact about the state of technology today that even good programs sometimes go wrong. It is not uncommon for programs to crash or misbehave, whether accidently, or with malicious intent. This problem has been greatly compounded in recent years by the proliferation of *mobile* code. More and more of the code that is run is downloaded in bits and pieces from various sources. Examples of this include Javascript and Java(tm) programs downloaded into a web browser, applications downloaded directly over the internet, and code run on behalf of others (such as the SETI@HOME project, which uses donated spare processor cycles to further the search for extra-terrestrial life). The proliferation of mobile code is expected only to increase as networked technology becomes more a part of everyday life.

Unfortunately, it is particularly hard to trust the behavior of this sort of code. The code producer may be unknown to the code consumer, or the identity of the producer may be spoofed. Moreover, even trusted producers occasionally produce programs that go wrong.

It would certainly seem desirable to be able to rule out programs that are unsafe. Unfortunately, determining the safety of arbitrary machine code is undecidable. A compromise solution that has been in existence for several decades, is to restrict ourselves to programming in a language which has the property that all programs are safe. The resulting compiled code is therefore safe by construction, *modulo the correctness of the compiler*. Languages such as LISP, ML and Java all have this property: that all programs written in these languages are guaranteed with some level of certainty not to crash.

Unfortunately for the purposes of mobile code, the safety properties enjoyed by these languages are only guaranteed at the source level: once the source code has been compiled to machine instructions, the safety guarantee lies only in the implicit property of being in the image of the safe language. One solution to this is to ship around instead something tantamount to source code, allowing the consumer to validate the code independently. This is in essence the Java(tm) byte-code solution. This places much of the burden of compilation on the code consumer, who must still in turn trust that their own compiler is correct.

The two most commonly used solutions then, are either to accept arbitrary machine code based on trust in the producer of the code, or to accept only annotated high-level code and to trust the local compiler. Both of these solutions leave much to be desired.

To better this, several systems have been proposed for annotating machine code in such a way that correctness remains *checkable*. Along with the code, the code consumer receives a certificate that can be used to *check* that that the code conforms to the correctness assertions that it claims. The only software that the code consumer must still trust is the checker itself. Two notable examples of this include Proof Carrying Code (PCC) from CMU [Nec98] and Typed Assembly Language (TAL) from Cornell [MWCG98]. The PCC system provides for machine code to be annotated with proofs of safety properties done in first-order logic. The code consumer simply checks that the proofs are indeed correct – a relatively simple procedure. This system is very general, in that it can be used to certify any property which can be expressed in the logic. TAL on the other hand specializes to the particular property of type safety. TAL provides for a machine code which is annotated with types, which can then be type checked by the code consumer.

A certifying compiler is one which produces, along with its normal output, a *certificate* which can be used to check that the generated code is safe according to some policy. Certifying compilers have been written translating safe subsets of C to both PCC and TAL [MCG$^+$99, NL98]. More ambitiously, a full scale Java(tm) compiler has been written targeting PCC [CLN$^+$00].

The **TILT** (TIL Two) compiler is an optimizing compiler developed at CMU that implements the full Standard ML '98 definition and includes support for separate compilation. Important ideas pioneered in **TILT** and its predecessor TIL include using intensional polymorphism [HM95] to reduce the cost of implementing polymorphism and garbage collection. Compilation proceeds as a series of typed transformations into successively lower level typed languages. Type information is used to allow for optimized data representations and to do "almost tag-free" garbage collection. Currently however, type information is mostly erased in **TILT** well before the transformation to machine code is made, and hence safety properties of the resulting code can only be asserted - not checked.

TILT uses types during compilation for optimization purposes, and consequently requires an intermediate language with a very expressive type theory. Previous work on typed assembly languages has primarily focused on preserving type information for certification purposes. I claim that these two uses of low level typed languages are compatible. *A compiler can use types to generate certified binaries while retaining the ability to perform complicated type based optimizations on a full modern language.*

I propose to demonstrate this by extending the TILT-ML compiler to maintain type information used in the intermediate passes of the compiler all the way through code generation, producing certified binaries without sacrificing the ability to perform type analysis optimizations. Section 2 describes the **TILT** compiler in more detail, and discusses some of the special challenges it presents. Section 3 describes at a high level the various aspects of my proposal. Section 4 gives an overview of previous and related work in this area. Section 5 provides a detailed look at my proposed translation from the **TILT** intermediate language to a low level typed assembly language.

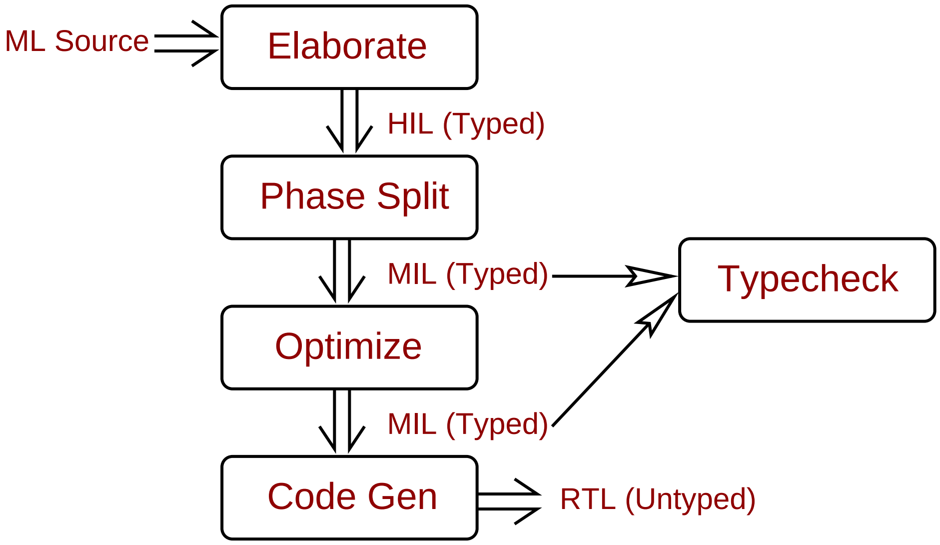

Figure 1: TILT Architecture

2 TILT

The past years have seen a great deal of interest in the idea of "typed compilation": that is, maintaining type information throughout the compilation process. This type information can be exploited by the compiler internally to allow for optimized data representations and to do tag-free garbage collection, as well as providing the compiler with a basis for internal correctness checks. This work was pioneered in the TIL compiler at CMU [TMC+96]. Other recent work has also suggested the possibility of maintaining type information through to the machine code as a form of certification [MWCG97].

The TIL compiler clearly demonstrated that typed compilation was both feasible and desirable. However, TIL compiled only the core language of Standard ML: the powerful modular features that are one of the most important elements of SML were not dealt with. The TIL Two (**TILT**) compiler was aimed at addressing this shortcoming.

Figure 1 depicts the structure of the **TILT** compiler. The **TILT** architecture is based around two typed intermediate languages. The initial elaboration from SML source targets a structures calculus called the HIL (High Intermediate Language). This language is relatively close to SML, and among other things provides the interface language used for separate compilation. After elaboration (and hence typechecking), the HIL is translated to a second typed language called the **MIL** (Middle Intermediate Language) through a process called phase splitting [HMM90]. The phase splitting process maps each SML structure into separate type and term level records, representing the static and dynamic portions of the structure. Similarly, SML functors are mapped to type and term level functions. In this fashion, modular programs are translated into programs containing only lambda calculus terms.

The **MIL** is the language in which almost all of the optimization passes implemented in **TILT** are done. This constrains the design of the **MIL**, since it must be possible to express the results all of the desired optimizations in a typed fashion. In particular, it is important that primitives for data representation optimizations be present at this level. By "hiding" type analysis inside of a few primitives, the **MIL** avoids the need for a general typecase construct as used in the λ_i^{ML} calculus. Nonetheless, the fact that some **MIL** primitives do indeed analyze their types mandates a type passing interpretation for the **MIL** operational semantics.

3 Plan of attack

This section describes at a high level the major pieces of my proposed work plan. These can be broken down as follows:

- Theory
- Producer side implementation (**TILT** compiler)
- Consumer side implementation (**TALT** typechecker, runtime)
- Evaluation

3.1 Theory

Section 5 describes the framework for a translation mapping the **TILT** internal language (the **MIL**) down to a typed assembly language that I call **TALT** (Typed Assembly Language for **TILT**). This translation uses a new internal language called the **LIL** (Low Level Language) as an intermediate stage. The **LIL** is an impredicatively typed lambda calculus based on Crary and Weirich's LX [CW99].

The **LIL** provides a very rich type system in which the type analysis of **TILT** can be represented using term level constructs. In addition to various engineering benefits, this fact allows us to take a type erasure interpretation, instead of the type passing interpretation apparently mandated by the **MIL** primitives. At the lowest level, real machines do not in fact have a notion of types as we know them. The fact that we can embed type analysis into the term level reflects in a typed fashion exactly the techniques already used in an untyped fashion for implementing type passing languages.

The translation from **MIL** to **LIL** makes type analysis explicit and allows us to represent closure converted code directly in the language without resorting to abstract closures as was the case in the **MIL**. I also describe a slightly lower level variant of the **LIL** in which memory allocation is made explicit via `malloc` primitives. Eventually I would like to extend the theory to allow the interaction with the garbage collector to be typed directly, but at the moment this interaction needs to be hidden behind allocation primitives.

Finally, I give a translation from the **LIL** into a typed assembly language called **TALT**. **TALT** code is essentially machine code with type annotations added allowing it to be typechecked. Currently, there are in addition to the standard assembly language instructions, several typed primitives corresponding to assembler macros. These primitives handle memory allocation (and hence the interaction with the garbage collector) and array bounds checking. One of the goals of the this work will be to minimize the number of the these primitives needed.

Other theoretical work left to be done is to give proofs of the soundness of the type systems of the respective low level languages, and of the correctness of the translations between them.

3.2 Compiler implementation

The first stage of the implementation work will be to implement the translation described above in **TILT**. Where the original architecture (see figure 1) switches to an untyped language, the new implementation must instead take the typed output and continue the compilation process in a typed setting, as shown in figure 2. To do this, the following compiler phases must be added:

- **MIL** singleton elimination
- Translation to **LIL**
- Closure conversion
- Translation to **TALT**

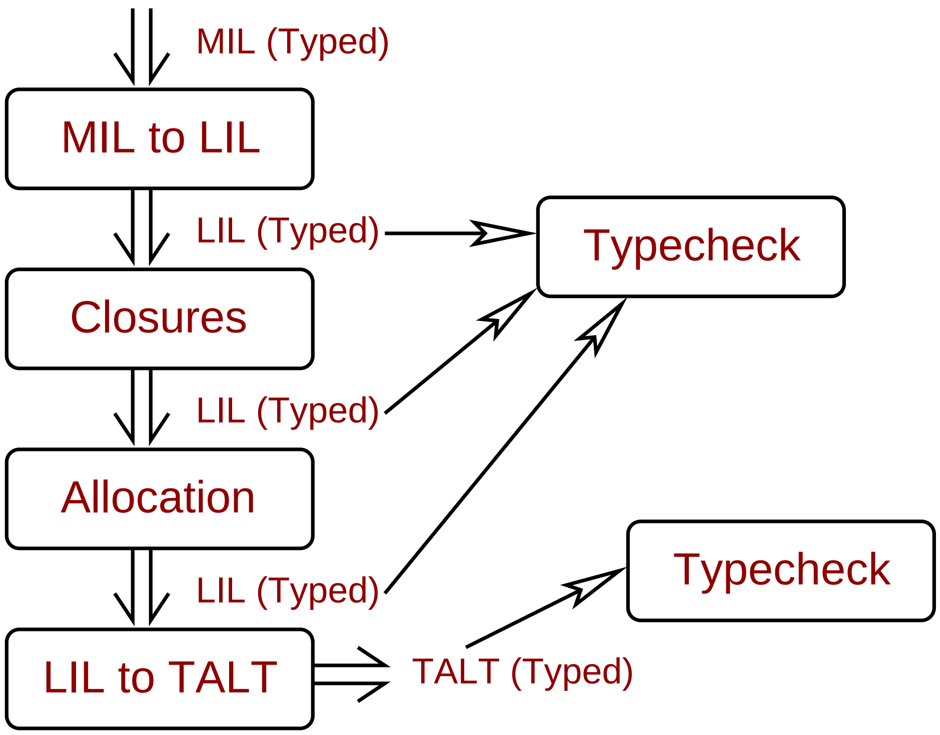

Figure 2: Planned Extension to the TILT Architecture

To simplify the meta-theory of the **LIL**, I have chosen not to support singleton kinds. Therefore, before (or concurrent with) the translation to **LIL**, singletons must be eliminated from **MIL** code. A proof that this is possible and a simple algorithm for doing so has been given by Crary [Cra00], but some work may be necessary to make a practical version of this that does not increase code size unnecessarily.

The translation to **LIL** is described in detail in section 5.3 and in appendix B. The most interesting part of this translation is the process of making type analysis explicit. The **MIL** has no general notion of type analysis, instead using type analyzing primitives to implement specific optimizations such as floating point array unboxing and flattening function arguments into records. The translation to **LIL** compiles these primitives into uses of a general type analysis mechanism. The **LIL** also provides constructs for making the allocation of memory explicit in the source language, though this will most likely be implemented as a separate pass.

The **MIL** implements all of the optimizations that **TILT** supports, so extended optimization of **LIL** code is presumably unnecessary. It would be appealing to rely as well on the **MIL** closure converter, and only translate closure converted code. Unfortunately, the **MIL** notion of closure conversion is not easily compatible with the **LIL** notion, since the **MIL** closure converts types. Translating closure converted **MIL** code would require "de-closure converting" types, which is not appealing. For this reason, a separate closure conversion pass for the **LIL** may be needed. Closure conversion in the **LIL** is discussed in more detail in section 5.4.

In order to allow intermediate code to be validated, it will also be necessary to implement a type checker for the **LIL** language. This will allow the output of the various phases to be checked for errors internally. The ability to check type correctness of internal program representations has proved valuable in the development of **TILT**.

3.3 TALT type checker implementation

The final piece of implementation work that needs to be done it to implement a typechecker for **TALT** itself. This is the consumer side of the code base, and the only piece of software that the consumer needs to trust. As such, the primary emphasis is on simplicity, but it is desirable that the typechecker not be unreasonably inefficient.

3.4 Evaluation

The final stage that I propose to undertake is a quantitative evaluation of the implementation. Perhaps most important is to measure the size of the generated code and certificates to get a feeling for the overhead of certification. Also important will be measurements of verification time. Less important but also interesting would be to measure the compilation time itself, and the quality of the generated code.

It is important to state here that while I propose to measure these quantities to understand the cost of my approach, optimizing for small certificates and fast certification time is beyond the scope of my proposal. The purpose of this work is to demonstrate the feasibility of using types to certify type analyzing code generated from a full-scale, general purpose language. Engineering the representations, while certainly important for making the compiler practical, is not a primary goal of this work.

3.5 Schedule

Phase	Allotted time
Theory	3 months
LIL typechecker	2 months
MIL to **LIL** translation	2 months
LIL Closure conversion	1 months
TALT typechecker and runtime	3 months
LIL to **TALT** translation	3 month
Measurements	1 month
Writing	4 months
Total	19 months

4 Related work

Greg Morrisett et al. described the original TAL language and used it as the target of a theoretical translation of System F [MWCG98]. This work was subsequently extended by the same authors to account for the use of an explicit control stack instead of the CPS compilation model of the original TAL paper [MCGW98]. Numerous other extensions have been made to TAL by various authors including support for dependent types [XH99], region based memory management [CWM99], and type safe linking [GM99]. A variant of TAL specialized to the x86 architecture was implemented by Morrisett et al. [MCG^{+}99], and targeted by a certifying compiler for a type safe variant of C [GM00].

The TAL model of certification is based on the idea of using type annotations to permit type safety properties to be checked by the code consumer. The Proof Carrying Code (PCC) system developed by George Necula at CMU uses an alternative strategy for certifying native code using first-order logic [Nec98]. PCC annotates binaries with proof terms which can be used by the code consumer to check that the code satisfies the desired safety properties. For certain properties (such as type safety) the proof terms can be automatically generated by the compiler, but the PCC framework has the ability to express and check more general properties that cannot be automatically inferred. Certifying compilers targeting PCC have been written for a small subset of ML [Nec97], and for a safe subset of C [NL98]. More recently, a full scale Java(tm) compiler has been written targeting PCC [CLN+00].

Appel and Felty [AF00] describe a "foundational" proof carrying code certification system based on higher-order logic. This system is expressive enough that the type-checking rules of the program can be proved as lemmas in the logic, so that the code consumer does not need to specify the type system as part of the safety policy. Appel [App01] describes ongoing work with Foundational PCC, including a new Typed Machine Language designed to serve as a target language for a certifying compiler.

5 Compiling MIL

In this section, I introduce a new intermediate language and give a translation into it from the existing **TILT** internal language (the **MIL**). I also describe a typed assembly language intended as the eventual target language of certified compilation in **TILT**, and informally describe compilation into it. The languages presented here are idealized and simplified from the actual internal languages. The primary intention of this translation is to demonstrate the means by which the type analysis in the **MIL** can be made explicit in a typed language that admits a type-erasure interpretation.

The presentation of the **TALT** target language is intended to suggest how the ideas used to implement type analysis in the **LIL** translate down to the assembly code level. The actual implementation is expected to depart significantly from that given here in terms of the operations and operands, if for no other reason than that the x86 architecture departs significantly from the idealized RISC architecture that this system is based on. I expect to formalize an x86-specific version of this system as part of my thesis work.

5.1 MIL

I describe the **MIL** here only briefly, as the theoretical and practical aspects of this calculus have been discussed in detail elsewhere [PCHS00, SH99]. For the purposes of the translation, I assume that singletons have already been eliminated [Cra00], although in practice it may be desirable to do this concurrently with the translation. The syntax of the singleton free **MIL** is given in figure 3. The **MIL** is a predicative lambda calculus based on Girard's F_ω, extended with primitives for type analysis. The intention is that these primitives are definable in terms of the λ_i^{ML} calculus of Harper and Morrisett [HM95]. In current practice, these primitives are only compiled directly after moving to an untyped calculus.

For the purposes of certification, this is unsatisfactory: I wish to maintain the types, and hence must give an accounting of the primitives which analyze them. The first step in this direction is the translation down to a language based on Crary and Weirich's LX [CW99] which I call the **LIL**, for Low Intermediate Language.

The syntax of the **MIL** language is given in figure 3. The kind structure is relatively simple in the absence of singletons, with function and product kinds to classify constructor functions and tuples, and the base kind T_{32} to classify constructors which classify terms. Most of the the base constructors are completely standard, with the exception of the treatment of sums and the constructs for type analysis. The sum type encodes the number of non value-carrying fields directly, instead of simply using unit as the carried value. In addition, the **MIL** has a known sum type, corresponding to the type of a sum for which the branch inhabited is known. A special projection operation projects the carried value out of a known sum. This allows the case construct to avoid destructing its value which may be unnecessary if the arm doesn't actually use the carried value.

Type analysis is present at the type level in the form of the `Vararg` construct and implicitly in the `Array` type. The `Vararg` type classifies the term-level `vararg` construct, which is used to implement non-standard calling conventions for functions which take tuples as arguments. In **TILT** tuple arguments to functions are always flattened into registers if the number of fields

$$
\begin{array}{lcl}
\kappa & ::= & \mathrm{T}_{32} \mid \kappa_1 \to \kappa_2 \mid \kappa_1 \times \kappa_2 \\
\\
c & ::= & \alpha \mid \mathtt{Int} \mid \mathtt{Boxf} \mid (\vec{c}) \to c \mid (\vec{c}) \to^t c \mid c_1 \times \ldots \times c_n \\
 & \mid & \mu(\alpha,\beta).(c_1,c_2) \mid \langle c_1, c_2 \rangle \mid \pi_1\, c \mid \pi_2\, c \mid \lambda(\alpha{::}\kappa).c \mid c_1 c_2 \\
 & \mid & \mathtt{Vararg}_{c_1 \to c_2} \mid \mathtt{Vararg}_{c_1 \to^t c_2} \mid \mathtt{Sum}_i(\vec{c}) \mid \mathtt{Sum}_i^j(\vec{c}) \\
 & \mid & \mathtt{Array}_c \mid \mathtt{Farray} \mid \mathtt{Exn} \mid \mathtt{Dyntag}_c \\
\\
\tau & ::= & T(c) \mid \forall[\vec{\alpha{::}\kappa}](\vec{\tau})(i) \to \tau \mid \forall[\vec{\alpha{::}\kappa}](\vec{\tau})(i) \to^t \tau \mid \tau_1 \times \ldots \times \tau_n \\
\\
\mathit{fv} & ::= & \mathfrak{r} \mid x_f \mid \mathtt{unboxf}\; \mathit{sv} \\
\\
\mathit{sv} & ::= & x \mid i \mid \mathtt{boxf}\; \mathit{fv} \mid \langle \vec{\mathit{sv}} \rangle \\
 & \mid & \mathtt{inj\_tag}^j_{\mathtt{Sum}_i(\vec{c})} \mid \mathtt{inj}^j_{\mathtt{Sum}_i(\vec{c})}\; \mathit{sv} \\
 & \mid & \mathtt{exn\_tag}_c \mid \mathtt{inj\_dyn}_c(\mathit{sv}_1, \mathit{sv}_2) \mid \mathtt{roll}_c\; \mathit{sv} \\
\\
\mathit{opr} & ::= & \mathit{sv} \mid \mathtt{unroll}_c\; \mathit{sv} \mid \mathtt{vararg}_{c_1 \to c_2}\; \mathit{sv} \mid \mathtt{onearg}_{c_1 \to c_2}\; \mathit{sv} \\
 & \mid & \mathtt{proj}^i\; \mathit{sv} \mid \mathtt{select}^i\; \mathit{sv} \\
 & \mid & \mathtt{case}(\mathit{sv})(x_1.e_1, \ldots, x_n.e_n) \mid \mathtt{handle}_\tau(e_1, x.e_2) \\
 & \mid & \mathtt{sub}_c(\mathit{sv}_1, \mathit{sv}_2) \mid \mathit{sv}[\vec{c}](\vec{\mathit{sv}})(\vec{\mathit{fv}}) \\
 & \mid & \mathtt{raise}_\tau\; \mathit{sv} \mid \mathtt{mkexntag}_c \\
 & \mid & \mathtt{exncase}(\mathit{sv})(\mathit{sv}_1 \Rightarrow x_1.e_1, \ldots, \mathit{sv}_n \Rightarrow x_n.e_n) \\
 & \mid & \mathit{fv} \mid \mathtt{fsub}(\mathit{sv}_1, \mathit{sv}_2) \\
 & \mid & \mathtt{array}_c(\mathit{sv}_1, \mathit{sv}_2) \mid \mathtt{farray}(\mathit{sv}, \mathit{fv}) \\
\\
e & ::= & \mathit{sv} \mid \mathtt{let}_\tau\; \mathtt{rec}_\tau\; f[\vec{\alpha{::}\kappa}](\vec{x{:}\tau})(\vec{x_f}).e\; \mathtt{in}\; e \\
 & \mid & \mathtt{let}_\tau\; x = \mathit{opr}\; \mathtt{in}\; e \mid \mathtt{let}_\tau\; \mathtt{upd}_c(\mathit{sv}_1, \mathit{sv}_2, \mathit{sv}_3)\; \mathtt{in}\; e \\
 & \mid & \mathtt{let}_\tau\; x_f = \mathit{opr}\; \mathtt{in}\; e \mid \mathtt{let}_\tau\; \mathtt{fupd}(\mathit{sv}_1, \mathit{sv}_2, \mathit{fv})\; \mathtt{in}\; e \\
\\
\Delta & ::= & \bullet \mid \Delta, \alpha{:}\kappa \mid \Delta, x{:}\tau \mid x_{64}
\end{array}
$$

Figure 3: **MIL** syntax

in the tuple is small. In order to make this work with polymorphic functions, it is necessary to use type analysis. The `Vararg` type can be thought of as simply a type level typecase in a λ_i^{ML} like language which branches on the argument type of a function type. If the argument type is a small tuple, it returns a multi-argument function type where the fields of the tuple have been flattened. Otherwise, it leaves the function type unchanged.

The `Array` type implicitly distinguishes between arrays of boxed floating point numbers and other arrays, in order to flatten the boxed float arrays. This is discussed in more detail below. The rest of the constructor level is a standard typed lambda calculus, classified by the function and product kinds.

Also described in figure 3 is the syntax for the type level. Unlike the constructor level which corresponds to the notion of *types as data*, the type level in a predicative system corresponds to the notion of *types as classifiers*. The constructor level is included into the type level via an explicit inclusion $T(c)$. The type level also contains classifiers for polymorphic functions and tuples of terms. The duplication of the the tuple type at the type level indicates the possibility of constructing pairs containing polymorphic functions which is not provided for by the constructor level.

For the most part, the term-level **MIL** is a standard lambda calculus with a few non-standard primitives. The presentation here restricts the syntax to a named form to reflect the form used internally in the **TILT** compiler. Unlike most lambda calculi, the MIL also includes low level data representation primitives (such as float boxing and unboxing primitives). This allows data representation optimizations to be expressed at the level of the **MIL**.

A key optimization that **TILT** implements is the use of non-uniform data representation. Many implementations of languages with polymorphism require that all values fit into a word. In particular, array elements must always be word-sized, which means that arrays of 64 bit floats (for example) must actually be arrays of pointers to floats. This is unfortunate because of the wasted space, the extra indirections to access the data, and because of the consequent loss of data locality. **TILT** avoids this by incorporating type analysis into the array primitives. By passing types at runtime and allowing code to dispatch on them, unboxed floating point arrays can be used with the appropriate subscript stride chosen at runtime.

The MIL calculus differs from the λ_i^{ML}calculus of [HM95] in that it does not contain an explicit type analysis construct such as typerec or typecase. This does not mean however that the idea of intensional type analysis has been abandoned: rather, the type analysis has been hidden inside the primitives which need to use it.

For example, **TILT** deals with floating point numbers by syntactically distinguishing between boxed and unboxed floats, with appropriate term level conversions between them. This allows the optimizer to deal directly with data representation optimizations, even at the relatively high level of the MIL. The syntactic restriction on unboxed floats prevents polymorphic functions from being instantiated with the unboxed float type, so that all polymorphic values take up 32 bits. All unboxed floating point arguments are segregated so that they may be passed in float registers. A special `Farray` type is provided corresponding to the type of arrays of unboxed floats, with corresponding term-level operations `farray`, `fsub` and `fupd`.

One obvious problem with this is that arrays of values whose type is not statically known to be `Float` would seem to have to use a boxed representation. By using type analysis in the array constructor as well as the subscript operator however, at least some of the difficulty can be avoided. Essentially, the `Array` constructor incorporates a typecase which selects the appropriate array representation based on the type of the carried values. Similarly, the term-level subscript and update operations must dispatch on the type to select the appropriate operation. In the case that the type turns out to be `Boxf`, the subscript operation will also be forced to rebox the float before returning it, since subscripting into an array of boxed floats returns a value of type `Boxf`.

Type analysis is also encoded into the `vararg` and `onearg` primitives which implement special calling conventions for single argument functions whose argument type is a small record type. For example, the term $\mathtt{vararg}_{c_1 \to c_2}\ sv$ corresponds roughly to the following code using an explicit

λ_i^{ML} style typecase:

```
typecase c1 of
    Record[]  => lambda[]. sv <>
  | Record[t] => lambda[x;t]. sv <x>
  | Record[t1,t2] => lambda[x1;t1,x2;t2]. sv <x1,x2>
  | _ => f
```

The onearg construct is the inverse of vararg, turning a variable argument function back to a normal function. The maximum number of fields that can be flattened (here shown as two) is a machine dependent parameter of the type theory.

The **MIL** hides type analysis by replacing certain stylized uses of typecase with primitives that analyze their type arguments. At some point however, it becomes necessary to make this analysis explicit. Currently in the compiler, this happens when the **MIL** is translated to a low-level untyped language. One of the major challenges in pushing type information down to the machine level in **TILT** is making this analysis explicit in a typed language that is amenable to translation to a typed assembly language. The next section discuss a strategy for doing this by reflecting type analysis into the term level of a more powerful lambda calculus.

5.2 LIL

There is an apparent disparity between the type-passing model of type analysis and the constructs available at the machine level. Evaluation of terms depends on the type arguments as well as the term arguments, but real machine processors are untyped. The ad-hoc solution used in **TILT** is to translate to an untyped setting, choosing term-level representatives for the type data in the process. Type analysis then becomes simple branches on the values chosen to represent various types. The problem with this approach is that there is no obvious type system for the resulting code. Consequently, code that uses types for certification cannot directly use this approach to certify completely compiled code.

In order to make this ad-hoc solution viable for certification, Crary, Weirich, and Morrisett describe a type theory that permits term-level representation of types and type analysis in a typed setting [CWM98] using primitive terms to represent types. Crary and Weirich subsequently extend this notion [CW99] to a type theory in which representation of types is definable using only the ordinary lambda calculus term constructs with an enriched type system. The **LIL** adopts this idea for its treatment of type analysis, and also extends the **MIL** with constructs for representing closures.

5.2.1 Constructors and Kinds

Figure 4 describes the syntax of the **LIL**. The most important change from the **MIL** is the enrichment of the kind structure: most importantly by the addition of sum kinds with corresponding introduction and elimination forms at the constructor level. Universal kinds ($\forall j.\kappa$) and inductive kinds ($\mu j.\kappa$) are also provided. Kind variables bound by inductive kinds are restricted to occur only positively. Rather than using a syntactic distinction to segregate out 64 bit floating point numbers as is done in the **MIL**, the **LIL** uses an explicit kind distinction, providing kinds T_{32} and T_{64} corresponding to the kind of the types of 32 and 64 bit expressions respectively.

The metavariables c, τ and ϕ are used to represent constructors of arbitrary kind, kind T_{32} and kind T_{64} respectively. The lambda calculus portion of the constructor level contains the usual introduction and elimination forms for sums, pairs, lambdas, unit, and kind abstraction. These constructs are entirely standard. Inductive kinds are introduced with a fold construct, $\mathtt{fold}_{\mu j.\kappa}\, c$, which injects a constructor c of kind $\kappa[\mu j.\kappa/j]$ into the kind $\mu j.\kappa$.

The elimination form for inductive kinds is one of the most complex constructs in the **LIL** and requires some explanation. The essential idea is to provide a form of primitive recursion over inductive kinds at the constructor level. A primitive recursive constructor $\mathtt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to$

$$
\begin{array}{lll}
\kappa & ::= & \mathrm{T}_{32} \mid \mathrm{T}_{64} \mid 1 \mid \kappa_1 \to \kappa_2 \mid \kappa_1 \times \kappa_2 \\
& & \mid \kappa_1 + \kappa_2 \mid j \mid \mu j.\kappa \mid \forall j.\kappa \\
\\
c, \tau, \phi & ::= & * \mid \alpha \mid \lambda\alpha{:}k.c \mid c_1 c_2 \\
& & \mid \langle c_1, c_2 \rangle \mid \pi_1\, c \mid \pi_2\, c \\
& & \mid \mathtt{inj}_1^{k_1+k_2}\, c \mid \mathtt{inj}_2^{k_1+k_2}\, c \mid \mathtt{case}(c, \alpha_1.c_1, \alpha_2.c_2) \\
& & \mid \mathtt{fold}_{\mu j.k}\, c \mid \mathtt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}(j \to \mu j.\kappa)\ \mathtt{in}\ c) \\
& & \mid \mathtt{Float} \mid \mathtt{Int} \mid \mathtt{Boxed} \mid \mathtt{Void} \mid \times \mid \to \\
& & \mid c[\kappa] \mid \Lambda j.c \mid \exists \mid \forall \mid \mathtt{Rec} \mid \mathtt{Sum} \mid \mathtt{KSum} \\
& & \mid \mathtt{Array}_{32} \mid \mathtt{Array}_{64} \mid \mathtt{Dyntag} \mid \mathtt{Dyn} \mid \mathtt{Tag} \\
\\
fv & ::= & x_{64} \mid \mathfrak{r} \\
sv & ::= & x \mid i \mid \mathtt{box}\, fv \mid \langle \vec{sv} \rangle \\
& & \mid \mathtt{inj\_tag}^{j}_{(i,c)} sv \mid \mathtt{inj}^{j}_{(i,c)}\, sv \mid \mathtt{inj\_dyn}_\tau(sv_1, sv_2) \mid \mathtt{roll}_\tau\, sv \\
& & \mid \mathtt{pack}\, sv\ \mathtt{as}\ \exists\alpha{:}\kappa.\tau\ \mathtt{hiding}\, c \mid sv[c] \\
& & \mid \mathtt{forgetknown}\, sv \mid \mathtt{tag}_i \\
\\
fopr & ::= & fv \mid \mathtt{sub}_\phi(sv_1, sv_2) \\
opr & ::= & sv \mid \mathtt{unroll}_c\, sv \mid \mathtt{select}^i\, sv \\
& & \mid \mathtt{case}(sv)(x.e_1, \ldots, x.e_n) \mid \mathtt{proj}^j\, sv \\
& & \mid \mathtt{dyncase}(sv)(sv_1 \Rightarrow x_1.e_1, \_ \Rightarrow e) \\
& & \mid \mathtt{dyntag}_\tau \mid \mathtt{raise}_\tau\, sv \mid \mathtt{handle}_\tau(e_1, x.e_2) \\
& & \mid \mathtt{unbox}\, sv \mid sv(\vec{sv})(\vec{fv}) \\
& & \mid \mathtt{vcase}_\tau(c, \alpha_1.\,\mathtt{dead}\, sv, \alpha_2.e) \mid \mathtt{vcase}_\tau(c, \alpha_1.e, \alpha_2.\,\mathtt{dead}\, sv) \\
& & \mid \mathtt{array}_\tau(sv_1, sv_2) \mid \mathtt{array}_\phi(sv, fv) \mid \mathtt{sub}_\tau(sv_1, sv_2) \\
& & \mid \mathtt{upd}_\phi(sv_1, sv_2, fv) \mid \mathtt{upd}_\tau(sv_1, sv_2, sv_3) \\
\\
e & ::= & \mathtt{let\,rec}_\tau\, f[\vec{\alpha{:}\kappa}](\vec{x{:}\tau})(\vec{x_{64}{:}\phi}).e\ \mathtt{in}\ e \\
& & \mid sv \mid \mathtt{let}_\tau\, x = opr\ \mathtt{in}\, e \mid \mathtt{let}\, x_{64} = fopr\ \mathtt{in}\, e \\
& & \mid \mathtt{let}\ [\alpha, x] = \mathtt{unpack}\, sv\ \mathtt{in}\, e \mid \mathtt{let}\ \langle \beta, \gamma \rangle = c\ \mathtt{in}\, e \\
& & \mid \mathtt{let}\ (\mathtt{fold}\, \beta) = c\ \mathtt{in}\, e \\
\\
\Delta & ::= & \bullet \mid \Delta, j \mid \Delta, \alpha{:}\kappa \\
\Gamma & ::= & \bullet \mid \Gamma, x{:}\tau \mid \Gamma, x_{64}{:}\phi
\end{array}
$$

Figure 4: **LIL** syntax

Constants and their kinds		
$\texttt{Float}{:}\mathrm{T}_{64}$	$\texttt{Int}{:}\mathrm{T}_{32}$	$\texttt{Void}{:}\mathrm{T}_{32}$
$\texttt{Array}_{32}{:}\mathrm{T}_{32} \to \mathrm{T}_{32}$	$\texttt{Array}_{64}{:}\mathrm{T}_{64} \to \mathrm{T}_{32}$	$\texttt{Boxed}\,{:}\mathrm{T}_{64} \to \mathrm{T}_{32}$
$\texttt{Tag}\,{:}\texttt{nat} \to \mathrm{T}_{32}$	$\texttt{Dyntag}\,{:}\mathrm{T}_{32} \to \mathrm{T}_{32}$	$\texttt{Dyn}\,{:}\mathrm{T}_{32}$
$\times{:}\mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{32}$	$\to{:}\mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{64}\texttt{list} \to \mathrm{T}_{32} \to \mathrm{T}_{32}$	
$\texttt{Sum}\,{:}\texttt{nat} \to \mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{32}$	$\texttt{KSum}\,{:}\texttt{nat} \to \texttt{nat} \to \mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{32}$	
$\forall{:}\forall j.(j \to \mathrm{T}_{32}) \to \mathrm{T}_{32}$	$\exists{:}\forall j.(j \to \mathrm{T}_{32}) \to \mathrm{T}_{32}$	
$\texttt{Rec}\,{:}\forall j.((j \to \mathrm{T}_{32}) \to (j \to \mathrm{T}_{32})) \to j \to \mathrm{T}_{32}$		

Figure 5: Constructor constants and their kinds

$\kappa'), i{:}(j \to \mu j.\kappa).c)$ recursively defines a function from $\mu j.\kappa$ to $\kappa'[\mu j.\kappa/j]$, with the body of the function give by c. The variable α is the argument to the function, and stands for the unfolded argument (nominally of kind $\kappa[\mu j.\kappa/j]$). However, in order to ensure that the function is only recursively called on a subcomponent of the argument (and hence is guaranteed to terminate), α is bound at kind κ with occurrences of j left abstract, and the recursive variable ρ always has j as its domain kind. The last bound variable of the `pr` construct (ι above) corresponds to a fold at the domain kind of the function. While probably not strictly necessary, this allows the original argument to be returned without traversing and rebuilding the entire structure.

The constructors classifying the term level are presented in the **LIL** as constants of higher kind. These constructors, given in figure 5, include constants for impredicative universal and existential types, general parameterized recursive types, arrays, tagged values, sums, integers, floating point numbers, boxed 64 bit values, pairs, and functions. Formulating these constructors as higher order constants makes the interaction with type analysis easier to deal with. The kinds of several of the constants refer to the kind of lists of constructors. This is defined as $\texttt{list}[k] = \mu j'.1 + k \times j'$. Throughout this proposal I will frequently use the usual ML list syntax for constructor lists. The kind `nat` describing the encoding of natural numbers is definable directly in the **LIL**in the usual fashion, and is used in the typing rules as well. When n is a natural number, I write $\overline{n}$ for the constructor of kind `nat` representing n.

For the most part, the constructor constants are fairly straightforward. For example, $\times c$ corresponds to the type of a tuple, the types of the fields of which are given by the elements of c. If $c = [\tau_1, \tau_2]$, then this corresponds to $\tau_1 \times \tau_2$ in a more standard notation. Similarly, the arguments to the $\to$ constructor correspond to the types of the 32 bit and 64 bit arguments of the function, and its return type. Hence, $\to([\tau])([\phi])(\tau)$ corresponds to $(\tau; \phi) \to \tau$. The `Sum` and `KSum` constructors classify sums and known sums. The arguments to `Sum` give the number of non-value carrying arms of the sum, and a list of the types in the value carrying arms. For example, the **LIL** version of $1 + \tau$ is $\texttt{Sum}(\overline{1})([\tau])$. The `KSum` constructor works similarly, but takes an additional argument giving the index of the inhabited arm of the sum. The `Tag` constructor classifies sum tags, and takes as its argument a constructor of kind `nat` indicating which branch of the sum the value inhabits.

The universal and existential constructors take as arguments a kind indicating the domain of quantification and and a constructor function giving the body. The standard $\forall(\alpha{:}\kappa).c$ becomes $\forall[\kappa](\lambda(\alpha{::}\kappa).c)$, and similarly for the existential. Finally, parameterized recursive types are defined with the `Rec` constructor. $\texttt{Rec}[\kappa](\lambda(\rho{:}\kappa \to \mathrm{T}_{32}).\lambda(\alpha{:}\kappa).c)(c')$ (where c' has kind κ) corresponds to recursively defining a constructor of kind $\kappa \to \mathrm{T}_{32}$ and applying it to c'. Within the body of c, ρ stands for the recursively defined constructor, and α stands for the parameter. Therefore, the unfolding of such a type is given by $c_r(\lambda(\alpha{:}\kappa).(\texttt{Rec}[\kappa](c_r)(\alpha)))(c')$, where $c_r = (\lambda(\rho{:}\kappa \to \mathrm{T}_{32}).\lambda(\alpha{:}\kappa).c)$.

5.2.2 Expressions

The expression level of the **LIL** is divided into four syntactic classes: 64 bit values (*fv*), 32 bit values (*sv*), operations (*opr*) and expressions (e). Programs are syntactically restricted to a named form where all intermediate computations are named in the usual fashion.

Values include variables, constants, tuples, polymorphic instantiation, and existential introduction. The `box` construct wraps a 64 bit value as a 32 bit value, corresponding eventually to heap allocation. Sum tags $\mathtt{tag}_i$ are made explicit, and the sum injections $\mathtt{inj\_tag}^j_{(i,c)}$ and $\mathtt{inj}^j_{(i,c)}$ are introduced to coerce values into known sums – the former coercing a tag, and the latter a tagged value. The `forgetknown` construct coerces a known sum to an unknown sum. These new sum constructs allow sum tag manipulation to be made explicit, unlike in the **MIL**. It also permits allocation of sums to be reduced to record allocation which simplifies later phases.

Operations are computations that return values, and are bound to variables within expressions. For simplicity, I include values into the operations to unify let binding into a single mechanism. Other operations include unrolling of recursive types, tuple selection, sum case, known sum projection, exception constructs, and unboxing of 64 bit values. Array operations are provided for both 32 and 64 bit arrays.

The most interesting operation is the `vcase`, or "virtual case" construct. Crary and Weirich [CW99] describe a system for implementing type analysis as case analysis on constructor-level sums, essentially defining typecase in terms of more standard type theoretic constructs. They also describe a variant of their system which allows for a type erasure interpretation but still supports type analysis as a programming idiom. In this variant, term-level sums are used to stand for analyzable constructors. The `vcase` construct provides a mechanism whereby information about the identity of types encoded as terms can be reflected back into the type level.

To understand how this works, consider a value v of type $\mathtt{case}(\alpha, \beta_1.c, \beta_2.\mathtt{Void})$. According to the type given, v is either of type c or of type `Void` depending on whether α gets instantiated with a left or a right injection. But since there are no closed values of type `Void`, it is apparent that α can not be instantiated with a right injection, since to do so would imply that v has type `Void`. In essence, v serves as a witness to the fact that α will be instantiated with the left injection.

This fact can be propagated back into the type system by using `vcase`. When the argument to `vcase` is a variable, its arms are type-checked with the variable replaced with the appropriate injection in the context and in the body of the arm (that is, in the left arm, the variable is replaced with the left injection, and similarly with the right). Therefore, in the second branch of the expression $\mathtt{vcase}(\alpha, \beta_1.e, \beta_2.\mathtt{dead}\,v)$ the value v has type `Void`, which implies that this branch cannot be reached and is dead code. Within the body of e, α is known to be the left injection, and v is known to have type c. When types are erased, it is sound to erase the `vcase` as well, since one arm is known to be dead code. The `vcase` construct is the key to implementing type analysis, as the subsequent section will show.

Expressions in the **LIL** are either small values, or let bindings. Recursive function binding, 32 bit variable binding, and 64 bit variable binding are provided. Existential unpacking is also included at this level. In addition, two special binding forms exist for refining constructor paths into variables for analysis. When α has kind $\kappa_1 \times \kappa_2$, the expression $\mathtt{let}\langle\beta, \gamma\rangle = \alpha\,\mathtt{in}\,e$ replaces all occurrences of α in e with the pair $\langle\beta, \gamma\rangle$. This means that projections from α can be turned into variables. The `vcase` construct can only refine the type of its argument when the argument is a variable: this construct allows this to be extended to paths as well. The $\mathtt{let}(\mathtt{fold}\,\beta) = \alpha\,\mathtt{in}\,e$ expression serves a similar purpose when α has a recursive kind.

The complete static semantics of the **LIL** is given in appendix A. Many of the constructs described here are also discussed in detail by Crary and Weirich [CW99].

5.2.3 Example: Floating point array flattening

The use of the **LIL** type refinement operations discussed above is somewhat non-intuitive. Before attempting to give a full account of the translation of the **MIL**, it may be useful to consider a small example demonstrating the implementation of a simple type-analysis optimization in the **LIL**. The optimization I will describe here, floating point array flattening, is one of several implemented by the **TILT** compiler. In this scheme, an array of boxed floating point numbers is implemented not as an array of pointers, but as an array of the actual 64-bit values. Polymorphic array operations dispatch on the type of the array contents to determine which primitives to use. In an informal λ_i^{ML} style notation, this corresponds to defining the following type:

$$\begin{aligned}\texttt{Array}_{\text{opt}}(\alpha) = \texttt{typecase}(\alpha)\ \texttt{of}\ & \texttt{Boxed}(\phi) \Rightarrow \texttt{Array}_{64}(\phi)\\ & \mid \_ \Rightarrow \texttt{Array}_{32}(\alpha)\end{aligned}$$

As an example, I will show how to write a term level array constructor $\texttt{array}_{\text{opt}}$ that can be used to construct such arrays.

Type analysis in the **LIL** is based on the idea of case analyzing abstract syntax trees representing types. At the kind level, sums and recursive kinds are used to build a *static representation* of types. For the purposes of this optimization, this static representation can be very simple: we need only to be able to distinguish between boxed 64-bit types and other types.

$$\mathrm{T}_{\text{opt}} \stackrel{\text{def}}{=} \mathrm{T}_{64} + \mathrm{T}_{32} \qquad \text{(* Boxed of } (\phi) \mid \text{Other of } (\tau)\text{*)}$$

The T_{opt} kind classifies these static representations. The comment is meant to suggest the intuitive correspondence between this definition and an ML-style representation of an abstract syntax tree.

Constructors of kind T_{opt} serve as representations of types. In order to produce the actual type being represented, an *interpretation* function mapping from static representations to the represented type is needed.

$$\begin{aligned}\text{interp} : \mathrm{T}_{\text{opt}} \to \mathrm{T}_{32} \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{\text{opt}}).\,\texttt{case}\,\alpha\ \texttt{of}\ & \texttt{inj}_1\,\beta \Rightarrow \texttt{Boxed}(\beta)\\ & \mid \texttt{inj}_2\,\beta \Rightarrow \beta\end{aligned}$$

The interp function computes the actual type based on the static representation passed to it at runtime (in a type passing semantics).

To be concrete, consider the static representations of boxed 64-bit floats and 32-bit ints:

$$\begin{aligned}C_{BF} &: \mathrm{T}_{\text{opt}} \stackrel{\text{def}}{=} \texttt{inj}_1^{\mathrm{T}_{\text{opt}}}\,\texttt{Float}\\ C_I &: \mathrm{T}_{\text{opt}} \stackrel{\text{def}}{=} \texttt{inj}_2^{\mathrm{T}_{\text{opt}}}\,\texttt{Int}\end{aligned}$$

Notice that the interpretation function maps each of these back to the original type. Also notice that this representation scheme does not guarantee that all boxed floating point types will be represented as a first injection: it is perfectly possible to stuff Boxed(Float) into the second arm of the sum. This is not a problem, since it merely causes arrays of such types to be represented in un-flattened form. In fact, one could look at this as a feature, since the optimization does not force you to choose one representation of the other.

The actual type of specialized arrays that was given informally above can now be written down directly.

$$\begin{aligned}\texttt{Array}_{\text{opt}}{:}\mathrm{T}_{\text{opt}} \to \mathrm{T}_{32} \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{\text{opt}}).\,\texttt{case}\,\alpha\ \texttt{of}\ & \texttt{inj}_1\ \beta \Rightarrow \texttt{Array}_{64}(\beta)\\ & \mid \texttt{inj}_2\ \beta \Rightarrow \texttt{Array}_{32}(\beta)\end{aligned}$$

The $\texttt{Array}_{\text{opt}}$ constructor takes as an argument the static representation of a type an case analyzes this representation to determine the appropriate type of array to use. When the representations are statically known, this type can be reduced directly to the underlying array type.

$$\begin{array}{lcl} \texttt{Array}_{\text{opt}}(C_{BF}) & \equiv & \texttt{Array}_{64}(\texttt{Float}) \\ \texttt{Array}_{\text{opt}}(C_{I}) & \equiv & \texttt{Array}_{32}(\texttt{Int}) \end{array}$$

Static representations then account for type analysis at the type level by representing types as constructor level abstract syntax trees. Type analysis at the term level is accounted for in an analogous fashion. Analyzable types are given *dynamic representations* at the term level which again encode the structure of types in a case-analyzable form.

The type of the dynamic representation of a type whose static representation is c is given by $R(c)$, where R is defined as follows:

$$\begin{array}{l} R : \mathrm{T}_{\text{opt}} \to \mathrm{T}_{32} \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{\text{opt}}).(\texttt{case}\,\alpha\,\texttt{of}\,\texttt{inj}_1\,\beta \Rightarrow \texttt{Unit} \mid \texttt{inj}_2\,\beta \Rightarrow \texttt{Void}) \\ \qquad\qquad\qquad\qquad\qquad + (\texttt{case}\,\alpha\,\texttt{of}\,\texttt{inj}_1\,\beta \Rightarrow \texttt{Void} \mid \texttt{inj}_2\,\beta \Rightarrow \texttt{Unit}) \end{array}$$

As discussed in the previous section, each arm of the sum here serves as a witness to the identity of α. A closed value of type $\texttt{case}\,c\,\texttt{of}\,\texttt{inj}_1\,\beta \Rightarrow \texttt{Unit} \mid \texttt{inj}_2\,\beta \Rightarrow \texttt{Void}$. can only be unit, since there are no closed values of type `Void`. The existence of such a value together with the correctness of the type system implies that c can only be a left injection. The `vcase` construct can be used to reflect this fact back into the type system.

The dynamic representations of a type are terms whose type are given by the operation of R on their dynamic representation. So for example, the dynamic representations of boxed 64-bit floats and of 32-bit ints are given as follows:

$$\begin{array}{lll} V_{BF} & :R(C_{BF}) & \stackrel{\text{def}}{=} & \texttt{inj}_1 * \\ V_I & :R(C_I) & \stackrel{\text{def}}{=} & \texttt{inj}_2 * \end{array}$$

To re-cap, static representations of types (such as C_{BF}) provide type level representations of the abstract syntax of the analyzable types. Dynamic representations of types (such as V_{BF}) provide term level representations of the abstract syntax of the analyzable types. The last step remaining is to show how to use these representations to implement type analysis.

Functions which analyze types (given in the **MIL** by special primitives) simply become functions which take as arguments the static and dynamic representations of the analyzable types and dispatch on them appropriately. The type of the function to construct flattened arrays reflects this:

$$\texttt{array}_{\text{opt}} : \forall(\alpha{:}\mathrm{T}_{\text{opt}}).R(\alpha) \to \texttt{Int} \to \text{interp}(\alpha) \to \texttt{Array}_{\text{opt}}(\alpha)$$

The type argument and the first term argument correspond to the static and dynamic representations of the type of the contents of the array. The implementation of the function simply case analyzes the dynamic representation to determine the appropriate primitive array operation to use.

$$\begin{array}{l} \texttt{array}_{\text{opt}} \stackrel{\text{def}}{=} \Lambda(\alpha{:}\mathrm{T}_{\text{opt}}).\lambda(x_\alpha{:}R(\alpha)).\lambda(i{:}\texttt{Int}).\lambda(\mathbf{y}{:}\,\text{interp}(\alpha)). \\ \quad \texttt{case}\,x_\alpha \\ \qquad \texttt{of}\;\texttt{inj}_1\,\mathbf{r} \Rightarrow \texttt{vcase}\,\alpha\;\;\texttt{of}\;\texttt{inj}_1\,\beta_1 \Rightarrow \texttt{array}_{64}[\beta_1](i, \texttt{unbox}(\mathbf{y})) \\ \qquad\qquad\qquad\qquad\qquad\qquad\;\; \mid \texttt{inj}_2\,\beta_2 \Rightarrow \texttt{dead}\,\mathbf{r} \\ \qquad\;\; \mid \texttt{inj}_2\,r \Rightarrow \texttt{vcase}\,\alpha\;\;\texttt{of}\;\texttt{inj}_1\,\beta_1 \Rightarrow \texttt{dead}\,r \\ \qquad\qquad\qquad\qquad\qquad\qquad\;\; \mid \texttt{inj}_2\,\beta_2 \Rightarrow \texttt{array}_{32}[\beta_2](i, y) \end{array}$$

The type argument α has the kind of represented types, T_{opt}. The term argument x_α is the dynamic representation of the type represented by α: it's type is given by $R(\alpha)$. The actual type of the contents of the array can only be referred to via the interp function, since it is unknown at compile time. Therefore, the y argument with which the array is to be initialized is given type interp(α). Similarly, the return type $\texttt{Array}_{\text{opt}}(\alpha)$ gives the type of the returned array as a function of what α turns out to be.

The key to understanding how this works is to observe the effect that the vcase constructs have on the types of the variables in the function. Consider just the first branch of the case analysis on x_α, where the body of the arm does a virtual case analysis of α. According to the typing rule for vcase, the second arm of the vcase will be type-checked with $\texttt{inj}_2\,\beta_2$ substituted everywhere for α, *including in the context.* This means that whereas outside the arm the variable r has type

$$\texttt{case}\,\alpha\;\texttt{of}\;\texttt{inj}_1\beta \Rightarrow \texttt{Unit} \mid \texttt{inj}_2\beta \Rightarrow \texttt{Void}$$

within the arm it has type

$$\texttt{case}\;\texttt{inj}_2\beta_2\;\texttt{of}\;\texttt{inj}_1\beta \Rightarrow \texttt{Unit} \mid \texttt{inj}_2\beta \Rightarrow \texttt{Void}$$

which is equivalent to simply Void. This satisfies the typeing rule for vcase, which requires the dead branch to exhibit a value of type Void as proof that the branch is in fact dead.

To illustrate this further, the following table shows the types for the variables r and y inside and outside of the first vcase, at the occurrences indicated in bold in the definition above.

Outside of vcase	Inside of vcase
y: $\texttt{case}\,\alpha\;\texttt{of}\;\;\texttt{inj}_1\,\beta \Rightarrow \texttt{Boxed}(\beta) \mid \texttt{inj}_2\,\beta \Rightarrow \beta$	**y**: $\texttt{Boxed}(\beta_1)$
r: $\texttt{case}\,\alpha\;\texttt{of}\;\texttt{inj}_1\beta \Rightarrow \texttt{Unit} \mid \texttt{inj}_2\beta \Rightarrow \texttt{Void}$	**r**:Void

As an exercise, the reader may verify that when called with the appropriate arguments, the optimized array function defined above does in fact reduce to the appropriate 32 or 64-bit array primitive based on the representation of the type chosen.

$$\begin{array}{ll} \texttt{array}_{\text{opt}}[C_{BF}](V_{BF})(10)(0.0) & \mapsto^* \texttt{array}_{64}[\texttt{Float}](10, 0.0) \\ \texttt{array}_{\text{opt}}[C_I](V_I)(10)(0) & \mapsto^* \texttt{array}_{32}[\texttt{Int}](10, 0) \end{array}$$

5.3 The MIL to LIL translation

The translation from the **MIL** into the **LIL** language is primarily interesting in that it makes the uses of type analysis in the **MIL** primitives explicit and adds representations for types so that this analysis can be done at the term level. This implements in a typed fashion what is done currently in an untyped setting.

5.3.1 The constructor and kind translation

.

The translation of the **MIL** constructors really makes use of two separate notions: the static representation of a constructor, and the dynamic representation of a constructor.

A constructor's static representation is a **LIL** constructor which encodes information about the **MIL** constructor which it represents. The static representation can be analyzed at the type level to determine what type it represents. It can also be translated by an object level interpretation function to determine the actual **LIL** type corresponding to the **MIL** type that it represents.

A constructor's dynamic representation on the other hand, is a **LIL** term which also encodes the same information about the original **MIL** constructor, but at the term level instead of at the constructor level. The dynamic representations are used to implement type dispatch at runtime.

The translation of **MIL** kinds into **LIL** kinds must take into account the static representation of constructors. **MIL** constructors of kind T_{32} are encoded by the translation to contain the information needed for type analysis. The type of such constructors is defined as follows.

$$\mathrm{T}_{\text{mil}} \stackrel{\text{def}}{=} \mathrm{T}_{32}\texttt{list} + 1 + \mathrm{T}_{32}$$

The type analysis primitives currently used in the **MIL** only need to distinguish between record types (with their fields), the boxed float type, and all other types. More powerful primitives can easily be encoded, simply by changing this definition.

Intuitively, $\mathrm{T}_{\mathrm{mil}}$ corresponds to an ML datatype as such:

$$\text{datatype } \mathrm{T}_{\mathrm{mil}} = \textbf{Record} \text{ of } \mathrm{T}_{32}\texttt{list} \mid \textbf{BFloat} \mid \textbf{Other} \text{ of } \mathrm{T}_{32}$$

For presentational purposes, I will often use ML pattern matching style notation using this intuitive correspondence.

The kind translation mapping **MIL** kinds to **LIL** kinds is defined to map the kind T_{32} to $\mathrm{T}_{\mathrm{mil}}$, and to leave all other kinds unchanged.

$$\begin{aligned}
|\mathrm{T}_{32}| &\stackrel{\text{def}}{=} \mathrm{T}_{\mathrm{mil}} \\
|\kappa_1 \to \kappa_2| &\stackrel{\text{def}}{=} |\kappa_1| \to |\kappa_2| \\
|\kappa_1 \times \kappa_2| &\stackrel{\text{def}}{=} |\kappa_1| \times |\kappa_2|
\end{aligned}$$

Constructors of kind $\mathrm{T}_{\mathrm{mil}}$ serve to represent **MIL** constructors for analysis purposes. For the purposes of classifying term-level data, it is necessary to construct the **LIL** type that corresponds to the **MIL** type being represented. This is done by defining an interpretation function interp of kind $\mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32}$.

$$\begin{aligned}
\text{interp} : \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32} \stackrel{\text{def}}{=} {} & \lambda(\alpha{:}\mathrm{T}_{\mathrm{mil}}). \\
& \quad \texttt{case}\,\alpha \\
& \qquad \textbf{Record}\, l \Rightarrow \times(l) \\
& \quad\ \ \mid \textbf{BFloat} \Rightarrow \texttt{Boxed}(\texttt{Float}) \\
& \quad\ \ \mid \textbf{Other}\, t \Rightarrow t
\end{aligned}$$

Notice that this function is an object level function as opposed to a meta-level translation. This is necessary, since type abstraction means that the interpretation cannot always be statically computed.

The actual constructor translation $|c|$ translates **MIL** constructors of kind κ to **LIL** constructors of kind $|\kappa|$. This means that constructors of kind T_{32} will be mapped to **LIL** constructors of kind $\mathrm{T}_{\mathrm{mil}}$. For clarity in the translation, I use the ML style notation suggested by the datatype given above. In fact, the constructors for this datatype can be written directly in the **LIL**.

$$\begin{aligned}
&\textbf{Other} && : \mathrm{T}_{32} \to \mathrm{T}_{\mathrm{mil}} && \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{32}).\, \texttt{inj}_2^{\mathrm{T}_{\mathrm{mil}}}(\texttt{inj}_2\,\alpha) \\
&\textbf{BFloat} && : \mathrm{T}_{\mathrm{mil}} && \stackrel{\text{def}}{=} \texttt{inj}_2^{\mathrm{T}_{\mathrm{mil}}}(\texttt{inj}_1\,*) \\
&\textbf{Record} && : \mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{\mathrm{mil}} && \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{32}\texttt{list}).\, \texttt{inj}_1^{\mathrm{T}_{\mathrm{mil}}}\,\alpha
\end{aligned}$$

Constructors of higher kind are translated directly by recursively translating their subcomponents. All other constructors are encoded into the $\mathrm{T}_{\mathrm{mil}}$ kind. The complete constructor translation is given in figure 6. This process is mostly straightforward with the exception of the translation of the **MIL** constructors which incorporate type analysis.

Because of the special treatment of arrays of boxed floating point numbers, the `Array` type needs to analyze the type of values the array contains. This is implemented by translating the `Array` type into a sum switch on the type. If the type being encoded turns out at run time to be a boxed floating point number, then the case statement will return the 64 bit array type: otherwise, it returns the 32 bit version.

$$\begin{aligned}
|\texttt{Array}_c| \stackrel{\text{def}}{=} \textbf{Other}(&\texttt{case}\,|c| \\
&\ \textbf{Record}\ \alpha \Rightarrow \texttt{Array}_{32}(\times(\alpha)) \\
&\ \textbf{BFloat} \Rightarrow \texttt{Array}_{64}(\texttt{Float}) \\
&\ \textbf{Other}\ t \Rightarrow \texttt{Array}_{32} t)
\end{aligned}$$

$$
\begin{array}{lcl}
|\alpha| & \stackrel{\text{def}}{=} & \alpha \\
|\lambda(\alpha{:}\kappa).c| & \stackrel{\text{def}}{=} & \lambda(\alpha{:}|\kappa|).|c| \\
|c_1 c_2| & \stackrel{\text{def}}{=} & |c_1||c_2| \\
|\,\pi_1\, c| & \stackrel{\text{def}}{=} & \pi_1\, |c| \\
|\,\pi_2\, c| & \stackrel{\text{def}}{=} & \pi_2\, |c| \\
|\langle c1, c2\rangle| & \stackrel{\text{def}}{=} & \langle |c1|, |c2|\rangle \\
|\mathtt{Int}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\mathtt{Int}) \\
|\,\mathtt{Sum}_i(c_1,\ldots,c_n)| & \stackrel{\text{def}}{=} & \mathbf{Other}(\mathtt{Sum}(i)[\text{interp}\,|c_1|,\ldots,\text{interp}\,|c_n|]) \\
|\mathtt{Exn}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\exists(\alpha{::}\mathrm{T}_{32}).(\alpha \times (\mathtt{Dyntag}\,\alpha))) \\
|\,\mathtt{Dyntag}_c\,| & \stackrel{\text{def}}{=} & \mathbf{Other}(\mathtt{Dyntag}(\text{interp}\,|c|)) \\
|\mathtt{Farray}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\mathtt{Array}_{64}\mathtt{Float}) \\
|\mathtt{Boxf}| & \stackrel{\text{def}}{=} & \mathbf{BFloat} \\
|\mathtt{Unit}| & \stackrel{\text{def}}{=} & \mathbf{Record}[] \\
|c^1| & \stackrel{\text{def}}{=} & \mathbf{Record}[\text{interp}\,|c|] \\
|c_1 \times c_2]| & \stackrel{\text{def}}{=} & \mathbf{Record}[\text{interp}\,|c_1|, \text{interp}\,|c_2|] \\
|c_1 \times \ldots \times c_n| & \stackrel{\text{def}}{=} & \mathbf{Record}([\text{interp}\,|c_1|,\ldots,\text{interp}\,|c_n|]) \\
|(c_1,\ldots,c_n) \to c| & \stackrel{\text{def}}{=} & \mathbf{Other}((\text{interp}\,|c_1|,\ldots,\text{interp}\,|c_n|) \to \text{interp}\,|c|) \\
|\mathtt{Array}_c| & \stackrel{\text{def}}{=} & \mathbf{Other}(\mathtt{case}\,|c| \\
 & & \quad \mathbf{Record}\ \alpha \Rightarrow \mathtt{Array}_{32}(\times(\alpha)) \\
 & & \quad \mathbf{BFloat}\ \Rightarrow \mathtt{Array}_{64}(\mathtt{Float}) \\
 & & \quad \mathbf{Other}\ \ t \Rightarrow \mathtt{Array}_{32}t) \\
|\mathtt{Vararg}_{c_1 \to c_2}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\mathtt{case}\,|c_1| \\
 & & \quad \mathbf{Record}\,[] \Rightarrow () \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{Record}\,[t] \Rightarrow (t) \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{Record}\,[t_1,t_2] \Rightarrow (t_1,t_2) \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{Record}\ l \Rightarrow (\times(l)) \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{BFloat}\ \Rightarrow (\mathtt{Boxed\,Float}) \to \text{interp}\,|c_2|) \\
 & & \quad \mathbf{Other}\ \ t \Rightarrow (t) \to \text{interp}\,|c_2|) \\
|\mu(\alpha,\beta).(c_1,c_2)| & \stackrel{\text{def}}{=} &
\end{array}
$$

$$
\begin{array}{l}
\langle \mathbf{Other}(\mathtt{rec}_{1+1}\langle f, \mathtt{inj}_1 *\rangle), \mathbf{Other}(\mathtt{rec}_{1+1}\langle f, \mathtt{inj}_2 *\rangle)\rangle \\
\quad \text{where} f = \lambda(\rho{:}1+1 \to \mathrm{T}_{32}).\lambda(\omega.1+1). \\
\qquad \mathtt{case}\,\omega \\
\qquad\quad \mathtt{inj}_1\,\_ \Rightarrow \text{interp}(|c_1|[\rho(\mathtt{inj}_1 *)/\alpha, \rho(\mathtt{inj}_2 *)/\beta]) \\
\qquad\quad |\ \mathtt{inj}_2\,\_ \Rightarrow \text{interp}(|c_2|[\rho(\mathtt{inj}_1 *)/\alpha, \rho(\mathtt{inj}_2 *)/\beta])
\end{array}
$$

Figure 6: The constructor translation $|\cdot| : \forall\kappa.\kappa \to |\kappa|$

$\mathtt{Vararg}$ types similarly become dispatches over the argument types to select the appropriate function type. Throughout this translation, we assume that only records with fewer than 3 fields get flattened into registers, but in general the number of fields is a machine dependent parameter of the translation.

$$
\begin{array}{rl}
|\mathtt{Vararg}_{c_1 \to c_2}| \stackrel{\text{def}}{=} & \mathbf{Other}(\mathtt{case}\,|c_1| \\
& \quad \mathbf{Record}\,[] \Rightarrow () \to \text{interp}\,|c_2| \\
& \quad \mathbf{Record}\,[t] \Rightarrow (t) \to \text{interp}\,|c_2| \\
& \quad \mathbf{Record}\,[t_1, t_2] \Rightarrow (t_1, t_2) \to \text{interp}\,|c_2| \\
& \quad \mathbf{Record}\,l \Rightarrow (\times(l)) \to \text{interp}\,|c_2| \\
& \quad \mathbf{BFloat} \Rightarrow (\mathtt{Boxed\,Float}) \to \text{interp}\,|c_2|) \\
& \quad \mathbf{Other}\ t \Rightarrow (t) \to \text{interp}\,|c_2|)
\end{array}
$$

The translations given above, $|\kappa|$ and $|c|$, give the kind of the static representation of a constructor and the representation itself, respectively. The second element of the mapping from **MIL** to **LIL** is giving dynamic representations to constructors in order to permit the move from a type-passing interpretation to a type erasure interpretation.

The dynamic representation of a constructor c is given as $\downharpoonright c \downharpoonleft$, which I refer to as taking the "down" of the constructor. The down of a **MIL** constructor, $\downharpoonright c \downharpoonleft$, is its **LIL** dynamic representation. The type of the dynamic representation of a constructor is driven by its kind: constructors of pair kind get represented by terms of pair type, etc. However, the type of the dynamic representation also depends on the static representation in order to create the type of the representation of constructors of kind T_{32}. So the down of a **MIL** kind κ classifying c is given by the translation $\downharpoonright c{:}\kappa \downharpoonleft$: the type of $\downharpoonright c \downharpoonleft$.

$$
\begin{array}{lcl}
\downharpoonright c{:}\mathrm{T}_{32} \downharpoonleft & \stackrel{\text{def}}{=} & R(|c|) \\
\downharpoonright c{:}\kappa_1 \to \kappa_2 \downharpoonleft & \stackrel{\text{def}}{=} & \forall[\alpha{:}|\kappa_1|](\downharpoonright \alpha{:}\kappa_1 \downharpoonleft) \to \downharpoonright c\alpha{:}\kappa_2 \downharpoonleft \\
\downharpoonright c{:}\kappa_1 \times \kappa_2 \downharpoonleft & \stackrel{\text{def}}{=} & \downharpoonright \pi_1\, c{:}\kappa_1 \downharpoonleft \times \downharpoonright \pi_2\, c{:}\kappa_2 \downharpoonleft
\end{array}
$$

For constructors of pair kind, the dynamic representation will be a pair consisting of the dynamic representations of the sub-components. For constructors of function kind, the representation will be a polymorphic function parameterized over the static representation of the argument kind and mapping dynamic representations of constructors of the argument kind to dynamic representations of the return kind.

The subtlety of the translation arises when the constructor has kind T_{32}. Here what is required is something of the nature of the representation types of λ_r [CWM98]: the type of representation of a specific type. Unlike in λ_r, these types are not primitive here. Instead, they are programmed directly in the **LIL** in the following fashion.

$$
\begin{array}{rl}
R : \mathrm{T}_{\text{mil}} \to \mathrm{T}_{32} \stackrel{\text{def}}{=} & \lambda(\alpha{:}\mathrm{T}_{\text{mil}}). \\
& \quad \mathtt{case}\,\alpha \\
& \quad\quad (\mathbf{Record}\ \beta \Rightarrow \\
& \quad\quad\quad (\mathtt{case}\,\beta\,([] \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void}) \\
& \quad\quad\quad + \mathtt{case}\,\beta\,([\_], \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void}) \\
& \quad\quad\quad + \mathtt{case}\,\beta\,([\_,\_] \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void}) \\
& \quad\quad\quad + \mathtt{case}\,\beta\,(\_ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void}) \\
& \quad\quad \mid \_ \Rightarrow \mathtt{Void}) \\
& \quad + \mathtt{case}\,\alpha\,(\mathbf{BFloat} \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void}) \\
& \quad + \mathtt{case}\,\alpha\,(\mathbf{Other}\ \_ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})
\end{array}
$$

This definition is somewhat subtle. At the top level, the representations of types are always sums, indicating whether the type being represented is a record, boxed float, or an undistinguished type. The value carried in each branch of the sum serves as a witness to the identity of the original type. So for example in the boxed float case the carried value will have type

$\mathtt{case}\,\alpha\,(\mathbf{BFloat}\ \_ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})$. Since the Void type is uninhabited, having a value of this type means that α can only be the static representation of Boxed(Float). This information can be reflected back into the type system via the special vcase construct, whereby code can use this witness to refine α.

The actual term-level representations of constructors are fairly intuitive given the types above.

$$
\begin{array}{lcl}
\downharpoonleft\alpha\downharpoonright & \stackrel{\text{def}}{=} & x_\alpha \\
\downharpoonleft\lambda(\alpha{:}\kappa).c\downharpoonright & \stackrel{\text{def}}{=} & \lambda[\alpha{:}|\kappa|](x_\alpha{:}\ \downharpoonleft\alpha{:}\kappa\downharpoonright).\ \downharpoonleft c\downharpoonright \\
\downharpoonleft c_1 c_2\downharpoonright & \stackrel{\text{def}}{=} & \downharpoonleft c_1\downharpoonright\ [|c_2|](\downharpoonleft c_2\downharpoonright) \\
\downharpoonleft\pi_i\, c\downharpoonright & \stackrel{\text{def}}{=} & \mathtt{select}_i\ \downharpoonleft c\downharpoonright \\
\downharpoonleft\langle c1, c2\rangle\downharpoonright & \stackrel{\text{def}}{=} & \langle\downharpoonleft c_1\downharpoonright, \downharpoonleft c_2\downharpoonright\rangle \\
\downharpoonleft\mathtt{Unit}\downharpoonright & \stackrel{\text{def}}{=} & \mathtt{inj}_1^{R(|\mathtt{Unit}|)}(\mathtt{inj}_1\langle\rangle) \\
\downharpoonleft c^1\downharpoonright & \stackrel{\text{def}}{=} & \mathtt{inj}_1^{R(|c^1|)}(\mathtt{inj}_2(\mathtt{inj}_1\langle\rangle)) \\
\downharpoonleft c^2\downharpoonright & \stackrel{\text{def}}{=} & \mathtt{inj}_1^{R(|c^2|)}(\mathtt{inj}_2(\mathtt{inj}_2(\mathtt{inj}_1\langle\rangle))) \\
\downharpoonleft c^{n\geq 2}\downharpoonright & \stackrel{\text{def}}{=} & \mathtt{inj}_1^{R(|c^n|)}(\mathtt{inj}_2(\mathtt{inj}_2(\mathtt{inj}_2\langle\rangle))) \\
\downharpoonleft\mathtt{Boxf}\downharpoonright & \stackrel{\text{def}}{=} & \mathtt{inj}_2^{R(|\mathtt{Boxf}|)}\langle\rangle \\
\downharpoonleft c\ \mathtt{as}\ \mu(\alpha,\beta).(c_1, c_2)\downharpoonright & \stackrel{\text{def}}{=} & \langle\mathtt{inj}_3^{R(|\ \pi_1\ c|)}\langle\rangle, \mathtt{inj}_3^{R(|\ \pi_2\ c|)}\langle\rangle\rangle \\
\downharpoonleft c\ \mathit{otherwise}\downharpoonright & \stackrel{\text{def}}{=} & \mathtt{inj}_3^{R(|c|)}\langle\rangle
\end{array}
$$

Type functions become term functions which take both the static and dynamic representations of the argument and return the dynamic representation of the result. Applications are modified in the corresponding fashion. Type level pairs and projections map to term level pairs and projections. For constructors of kind T_{32}, it is necessary to construct the type representations by injecting the appropriate witnesses into the sum type described above.

These two mappings, $\downharpoonleft\cdot\downharpoonright$ and $|\cdot|$, complete the apparatus needed to translate proper **MIL** constructors. Constructors of kind T_{32} map to constructors of kind $\mathrm{T}_{\mathrm{mil}}$, which permits analysis of the form of the original constructor. However, since proper types are not analyzable in the **MIL** (only constructors of kind T_{32} can be analyzed) the translation of proper **MIL** types does not map into the $\mathrm{T}_{\mathrm{mil}}$ kind, but instead goes directly to kind T_{32}.

$$
\begin{array}{l}
|\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau| \quad \stackrel{\text{def}}{=} \\
\quad \forall[\alpha_1{:}|\kappa_1|, \ldots, \alpha_n{:}|\kappa_n|] \\
\qquad (\downharpoonleft\alpha_1{:}\kappa_1\downharpoonright, \ldots \downharpoonleft\alpha_n{:}\kappa_n\downharpoonright, |\tau_1|, \ldots, |\tau_m|)(\mathtt{Float}_0, \ldots, \mathtt{Float}_{k-1}) \to |\tau| \\
|T(c)| \quad \stackrel{\text{def}}{=} \quad \mathrm{interp}\ |c| \\
|\tau_1 \times \ldots \times \tau_n| \quad \stackrel{\text{def}}{=} \quad \times[|\tau_1|, \ldots, |\tau_n|]
\end{array}
$$

The most interesting piece of this translation is the treatment of polymorphic functions. Additional arguments corresponding to the dynamic representations of the type arguments are added to the parameter list of the function. Other types are translated compositionally. Notice though that the translation of a constructor used as a type is the *interpretation* of the translation of the constructor. This reflects the fact that proper **MIL** types are not available for analysis - they simply serve as classifiers.

The type and kind translations can be used to give a definition of the translation of a **MIL** typing context in the obvious manner.

$$
\begin{array}{lcll}
|\Delta, \alpha{:}\kappa| & \stackrel{\text{def}}{=} & \Delta', \alpha{:}|\kappa|; \Gamma & \text{if } |\Delta| = \Delta'; \Gamma \\
|\Delta, x{:}\tau| & \stackrel{\text{def}}{=} & \Delta'; \Gamma', x{:}|\tau| & \text{if } |\Delta| = \Delta'; \Gamma \\
|\Delta, x_{64}| & \stackrel{\text{def}}{=} & \Delta'; \Gamma', x{:}\mathtt{Float} & \text{if } |\Delta| = \Delta'; \Gamma
\end{array}
$$

In order to state properties of open terms, I also define the "down" of a typing context $\lfloor\Delta\rfloor$.

$$\begin{array}{rcl}\lfloor\Delta, \alpha{:}\kappa\rfloor & \stackrel{\text{def}}{=} & \lfloor\Delta\rfloor, x_\alpha{:}\ \lfloor\alpha{:}\kappa\rfloor \\ \lfloor\Delta, x{:}\tau\rfloor & \stackrel{\text{def}}{=} & \lfloor\Delta\rfloor \\ \lfloor\Delta, x_{64}\rfloor & \stackrel{\text{def}}{=} & \lfloor\Delta\rfloor\end{array}$$

Taking the down of a typing context introduces a new expression level binding for each type variable, corresponding to the dynamic representation of the type.

These two definitions provide enough mechanism to state the two principle soundness properties of the constructor and kind translations.

Conjecture 1 (Soundness of $|\cdot|$)
If $\Delta \vdash c : \kappa$ then $\Delta_l \vdash |c| : |\kappa|$ and if $\Delta \vdash \tau$ then $\Delta_l \vdash |\tau| : \mathrm{T}_{32}$, where $|\Delta| = \Delta_l; \Gamma_l$

Conjecture 2 (Soundness of $\lfloor\cdot\rfloor$)
if $\Delta \vdash c : \kappa$, then $|\Delta|, \lfloor\Delta\rfloor \vdash \lfloor c{:}\kappa\rfloor\ : \mathrm{T}_{32}$ and $|\Delta|, \lfloor\Delta\rfloor \vdash \lfloor c\rfloor\ :\ \lfloor c{:}\kappa\rfloor$.

5.3.2 The term translation

The translation from **MIL** terms to **LIL** terms is not notionally more complicated than the constructor translation, but is syntactically somewhat more cumbersome. The **MIL** allows subtyping of known and unknown sum types which must be eliminated in the translation to the **LIL**. Because of this, the term-level translation is a typed translation instead of purely syntactic. Moreover, the restriction to named form means that the translation of small values and operations may generate additional bindings that need to be lifted out. The translation of values and operations therefore must produce a list of bindings in addition to the translated term. The judgements used in the translation are as follows:

- $\Delta \vdash sv : \tau \leadsto \vec{d}\,\mathtt{in}\,sv'$
- $\Delta \vdash fv : \mathtt{Float} \leadsto \vec{d}\,\mathtt{in}\,fv'$
- $\Delta \vdash opr : \tau \leadsto \vec{d}\,\mathtt{in}\,opr'$
- $\Delta \vdash e : \tau \leadsto e'$

For convenience in the translation, I define a multiple let construct as follows:

$$\begin{array}{rcl}\mathtt{let}\,\epsilon\,\mathtt{in}\,e & \stackrel{\text{def}}{=} & e \\ \mathtt{let}\,\vec{d}, d\,\mathtt{in}\,e & \stackrel{\text{def}}{=} & \mathtt{let}\,\vec{d}\,\mathtt{in}\ \mathtt{let}\,d\,\mathtt{in}\,e\end{array}$$

Conjecture 3 describes the intended soundness properties of the translation judgements.

Conjecture 3 (Soundness of the term translation)

1. *if $\Delta \vdash sv : \tau \leadsto \vec{d}\,\mathtt{in}\,sv'$ then $|\Delta|, \lfloor\Delta\rfloor \vdash \mathtt{let}\,\vec{d}\,\mathtt{in}\,sv' : |\tau|$*
2. *if $\Delta \vdash fv : \mathtt{Float} \leadsto \vec{d}\,\mathtt{in}\,fv'$ then $|\Delta|, \lfloor\Delta\rfloor \vdash \mathtt{let}\,\vec{d}\,\mathtt{in}\ \mathtt{box}\,fv' : \mathtt{Boxed}\,\mathtt{Float}$*
3. *if $\Delta \vdash opr : \tau \leadsto \vec{d}\,\mathtt{in}\,opr'$ then $|\Delta|, \lfloor\Delta\rfloor \vdash \mathtt{let}\,\vec{d}, x = opr'\,\mathtt{in}\,x : |\tau|$, where x is fresh.*
4. *if $\Delta \vdash e : \tau \leadsto e'$ then $|\Delta|, \lfloor\Delta\rfloor \vdash e' : |\tau|$*

The complete **MIL** to **LIL** translation is given in appendix B, but some of the more interesting rules are discussed below. The translation judgments make use of a defined **typecase** construct on **MIL** constructors. The **typecase** construct branches on the top level structure of the type: that is, whether it is a record, a boxed float, or some other type. This is used in the translation for clarity in cases where the construct does not need to analyze the deeper structure of the type.

$$
\begin{array}{l}
\textbf{typecase}\, c\,(e_1, e_2, e_3) \stackrel{\text{def}}{=} \\
\quad \texttt{case}\, \downharpoonleft c \downharpoonright \\
\qquad \textbf{Record}\, \_ \Rightarrow \texttt{vcase}\, |c|\, (\textbf{Record}\, \_ \Rightarrow e_1 \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad \mid \textbf{BFloat}\, \_ \Rightarrow \texttt{vcase}\, |c|\, (\textbf{BFloat}\, \_ \Rightarrow e_2 \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad \mid \textbf{Other}\, \_ \Rightarrow \texttt{vcase}\, |c|\, (\textbf{Other}\, \_ \Rightarrow e_3 \mid \_ \Rightarrow \texttt{dead}\, x)
\end{array}
$$

Here the representation of the **MIL** constructor c will have type $R(\downharpoonleft c \downharpoonright)$, and consequently must be a term-level sum indicating the top level structure of the type. The appropriate branch of the **typecase** is thereby chosen via an ordinary case statement and evaluated. The `vcase` serves to refine the type argument to reflect the identity of the type back into the type system.

The first interesting case to consider is that of polymorphic functions. As suggested by the translation of function types, the translation of a polymorphic function binding must add additional term level variables to the argument list. These allow the dynamic representation of the type arguments to the function to be passed in.

$$
\frac{\begin{array}{c}
\Delta[f : \forall[\alpha \vec{::} \kappa](\vec{\tau})(|x_f|) \to \tau_r, \alpha \vec{::} \kappa, x \vec{:} \tau, \vec{x_f}] \vdash e_f : \tau_r \leadsto e'_f \\
\Delta[f : \forall[\alpha \vec{::} \kappa](\vec{\tau})(|x_f|) \to \tau_r] \vdash e : \tau \leadsto e'
\end{array}}{\begin{array}{l}
\Delta \vdash \texttt{let}_\tau\, \texttt{rec}_{\tau_r}\, f[\alpha \vec{::} \kappa](x \vec{:} \tau)(\vec{x_f}).e_f\, \texttt{in}\, e : \forall[\alpha \vec{::} \kappa](\vec{\tau})(|x_f|) \to \tau \leadsto \\
\qquad \texttt{let}_{|\tau|}\, \texttt{rec}_{|\tau_r|}\, f[\overrightarrow{\alpha{::}|\kappa|}](\overrightarrow{x_\alpha{:}\, \downharpoonleft \alpha{::}\kappa \downharpoonright}, \overrightarrow{x{:}\tau})(\overrightarrow{x_f{:}\texttt{Float}}).e'_f\, \texttt{in}\, e'
\end{array}}
$$

The additional type arguments to functions serve to make explicit the passing around of the information needed for type dispatch. The other interesting cases in the translation are the ones that actually dispatch on this information: in particular, arrays and variable argument functions.

Array construction branches on the type of the carried value to allow arrays of boxed floating point numbers to be flattened, as follows.

$$
\frac{\Delta \vdash sv_1 : \texttt{Int} \leadsto \vec{d_1}\, \texttt{in}\, sv'_1 \quad \Delta \vdash sv_2 : c \leadsto \vec{d_2}\, \texttt{in}\, sv'_2}{\begin{array}{l}
\Delta \vdash \texttt{array}_c(sv_1, sv_2) : \texttt{Array}_c \leadsto \\
\quad \vec{d_1}, \\
\quad \vec{d_2}, \\
\quad x = \textbf{typecase}\, c\, (\\
\qquad \texttt{array}_{\text{interp}\,|c|}(sv'_1, sv'_2), \\
\qquad \texttt{array}_{\texttt{Float}}(sv'_1, \texttt{unbox}\, sv'_2), \\
\qquad \texttt{array}_{\text{interp}\,|c|}(sv'_1, sv'_2)) \\
\quad \texttt{in}\, x
\end{array}}
$$

The subscript operation (and similarly for update) must parallel this by branching on the type and using the appropriate subscript.

$$
\frac{\Delta \vdash sv_1 : \texttt{Array}_c \leadsto \vec{d_1}\, \texttt{in}\, sv'_1 \quad \Delta \vdash sv_2 : \texttt{Int} \leadsto \vec{d_2}\, \texttt{in}\, sv'_2}{\begin{array}{l}
\Delta \vdash \texttt{sub}_c(sv_1, sv_2) : c \leadsto \vec{d_1}, \vec{d_2}, \\
\qquad x = \textbf{typecase}\, c\, (\\
\qquad\quad \texttt{sub}_{\text{interp}\,|c|}(sv'_1, sv'_2) \\
\qquad\quad \texttt{box}(\texttt{sub}_{\texttt{Float}}(sv'_1, sv'_2)) \\
\qquad\quad \texttt{sub}_{\text{interp}\,|c|}(sv'_1, sv'_2)) \\
\qquad \texttt{in}\, x
\end{array}}
$$

Notice that in the case that the array is a flattened float array, the result must be boxed again before returning it. If the type is statically known, both the branching and possibly the boxing can be optimized away, but in general they may need to be present.

For the specialized floating point array type and its corresponding primitives this optimization is built in, as can be seen in the translation rule for the specialized floating point subscript operation:

$$\frac{\Delta \vdash sv_1 : \texttt{Array}_c \rightsquigarrow \vec{d_1}\,\texttt{in}\, sv'_1 \quad \Delta \vdash sv_2 : \texttt{Int} \rightsquigarrow \vec{d_2}\,\texttt{in}\, sv'_2}{\Delta \vdash \texttt{fsub}(sv_1, sv_2) : \tau \rightsquigarrow \vec{d_1}, \vec{d_2}\ \texttt{in}\ \texttt{sub}_{\texttt{Float}}(sv'_1, sv'_2)}$$

The translation of the variable argument function primitives is slightly more complicated, in addition to dispatching on the top level form of the type of the argument, it must also distinguish between record types with different numbers of fields. In the case that the argument type is a record type, both vararg and onearg must consider the number of fields of the record to determine whether or not to emit a wrapper function, and what the argument types should be. For vararg, this wrapper function packages its arguments into a record and passes it on to the original function. The onearg construct simply reverses this.

$$\frac{\Delta \vdash sv : c_1 \to c_2 \rightsquigarrow \vec{d}\,\texttt{in}\, sv'}{\begin{array}{l}
\Delta \vdash \texttt{vararg}_{c_1 \to c_2}\, sv : \texttt{Vararg}_{c_1 \to c_2} \rightsquigarrow \\
\vec{d}, \\
f' = \texttt{case}\, \lfloor c_1 \rfloor\, (\\
\quad \textbf{Record}\, y \Rightarrow \texttt{vcase}\, |c_1|\, (\\
\quad\quad \textbf{Record}\, \alpha \Rightarrow \texttt{case}\, y \\
\quad\quad\quad |\ \textbf{R0}\, x \Rightarrow \texttt{vcase}\, \alpha \\
\quad\quad\quad\quad ([] \Rightarrow \texttt{let rec}\, f[]()().sv'[](\langle\rangle)()\,\texttt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad |\ \textbf{R1}\, x \Rightarrow \texttt{vcase}\, \alpha \\
\quad\quad\quad\quad ([\beta] \Rightarrow \texttt{let rec}\, f[](y{:}\beta)().sv'[](\langle y\rangle)()\,\texttt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad |\ \textbf{R2}\, x \Rightarrow \texttt{vcase}\, \alpha \\
\quad\quad\quad\quad ([\beta_1, \beta_2] \Rightarrow \texttt{let rec}\, f[](y_1{:}\beta_1, y_2{:}\beta_2)().sv'[](\langle y_1, y_2\rangle)()\,\texttt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad |\ \textbf{BigRecord}\, _ \Rightarrow sv' \\
\quad\quad |\ _ \Rightarrow \texttt{dead}\, y) \\
\quad |\ _ \Rightarrow sv') \\
\texttt{in}\, f'
\end{array}}$$

5.4 Closure converted LIL

The **MIL** language already implements closure conversion using abstract closures. Unfortunately, the type-passing implementation of type analysis used by the **MIL** means that in addition to closure converting the term level, the **MIL** also closure converts types. This makes the closure converted **MIL** code unsuited for translation to **LIL** code, since this would require the type level closures to be eliminated from the types. While this seems possible, it also seems unnecessarily complicated compared to simply translating pre-closure conversion **MIL** code and implementing closure conversion separately for the term level of the **LIL**.

Unlike the **MIL**, the **LIL** provides existential types which allows closures to be typed directly. Because the **LIL** admits a type erasure interpretation, polymorphic instantiation is a value. This is important, since closure conversion will instantiate polymorphic functions when closures are built.

For clarity, I define a minor syntactic variant of the **LIL** into which the closure conversion phase maps. This language, described in figure 7, reflects the fact that all closure converted code is closed and can be hoisted to the top level. The new typing rules for the closure converted **LIL** are given in appendix A.1.

In order to hoist functions to the top level, it is necessary to closure convert with respect to types as well as terms in a polymorphic setting. However, since I intend to erase types prior

$$
\begin{array}{lll}
k & ::= & \ldots \\
c, \tau, \phi & ::= & \ldots \\
fv & ::= & \ldots \\
sv & ::= & \ldots \\
opr & ::= & \ldots \\
fopr & ::= & \ldots \\
e & ::= & \boxed{\texttt{delete rec}_\tau\, f[\vec{\alpha{:}\kappa}](\vec{x{:}\tau})(\vec{y{:}\phi}).e} \mid \ldots \\
hval & ::= & \texttt{code}_\tau[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n](x_1{:}\tau, \ldots, x_m{:}\tau_m)(y_1{:}\phi_1, \ldots, y_k{:}\phi_k).e \\
d & ::= & f{:}\tau \mapsto hval \\
p & ::= & \texttt{letrec}\, \vec{d}\, \texttt{in}\, e
\end{array}
$$

Figure 7: Closure converted, named form, hoisted **LIL**

to execution, it is not necessary to include type environments in closures: instead I instantiate polymorphic functions before adding them to a closure. This transformation is completely straightforward and presents no new challenges [CWM98, MWCG97, MMH96, MH98].

5.5 LIL memory allocation

The second major transformation performed on the **LIL** is to make memory allocation explicit. This could in principle to be done as part of the translation to typed assembly language but it seems desirable to make this explicit at an earlier stage in order to expose the memory allocation primitives to optimization phases.

Figure 8 gives a final variant of the **LIL** which serves as the target of the memory allocation pass. At the value level tuples and boxed floating point numbers disappear, and labels ℓ corresponding to indices or pointers into the heap are introduced. At the instruction level, the 32 and 64 bit array creation constructs are replaced by corresponding allocation instructions of the form $\texttt{malloc}_\tau(sv, sv)$ and $\texttt{malloc}_\phi(sv, fv)$. The malloc form $\texttt{malloc}[\tau_1, \ldots, \tau_n]\langle sv_1, \ldots, sv_n\rangle$ is added for allocating tuples, and a $\texttt{box}\, fv$ instruction for allocating 64 bit values on the heap. Programs are wrapped in a heap which contains allocated data and code. Heap values are defined for arrays, tuples, floats, and functions.

The new rules for the static semantics of this variant of the **LIL** are given in appendix A.2. Notice that heap typing contexts giving types to labels are added to all of the term-level judgements. The judgment $\Psi \vdash hval : \tau$ **hval** describes well-formed heap values.

The memory allocation primitives described here are still relatively high level in that all of the allocation instructions require that initial values be given immediately as arguments. In general it is preferable to be able to make the initialization of memory explicit, both from the desire to expose the instructions to code moving optimizations and from the desire to be as close to the actual machine instructions as possible. The original TAL system handled initialization by putting flags on the types of allocated data to allow the type system to conservatively track the initialization of data. This strategy seem applicable to the the **LIL** as well. Part of my thesis research will be devoted to designing appropriate abstractions for dealing with memory allocation and the interface to the garbage collector. For the the purposes of this proposal however, I leave these notions abstract by hiding allocation and initialization within the `malloc` instructions.

$$
\begin{array}{lll}
k & ::= & \ldots \\
c, \tau, \phi & ::= & \ldots \\
sv & ::= & \ldots \boxed{\texttt{delete } \langle \vec{sv} \rangle,\ \texttt{box}\, fv} \\
 & & \mid \ell \\
 & & \\
fopr & ::= & \ldots \boxed{\texttt{delete array}_{\phi}(sv, fv)} \\
opr & ::= & \ldots \boxed{\texttt{delete array}_{\tau}(sv, sv)} \\
 & & \mid \texttt{malloc}[\tau_1, \ldots, \tau_n]\langle sv_1, \ldots, sv_n \rangle \mid \texttt{malloc}_{\tau}(sv_1, sv_2) \\
 & & \mid \texttt{box}\, fv \mid \texttt{malloc}_{\phi}(sv, fv) \\
 & & \\
hval & ::= & \langle \vec{sv} \rangle \mid [\vec{sv}] \mid fv \mid [\vec{fv}] \mid \texttt{exn\_tag} \\
 & & \mid \texttt{code}_{\tau}[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](x_1{:}\tau, \ldots, x_m{:}\tau_m)(\vec{x_f}).e \\
d & ::= & \ell{:}\tau \mapsto hval \\
p & ::= & \texttt{letrec}\, \vec{d}\, \texttt{in}\, e \\
\Psi & ::= & \bullet \mid \Psi, \ell{:}\tau
\end{array}
$$

Figure 8: **LIL** with memory allocation primitives

5.6 TALT

The final stage of compilation translates **LIL** programs to **TALT** code. This process makes the control structure of the program explicit via call, return, and jump instructions, and replaces variables with stack slots and registers. The actual code generation is not unusual except in that it preserves type information.

The syntax for **TALT** is described in figure 9. The type level adds constructs for describing the type of stacks and a kind ST classifying them. Function type are eliminated and replaced with a code type which describes the stack and register format expected by a code segment. "Nonsense" types ns_{32} and ns_{64} are introduced to describe uninitialized stacks slots, with corresponding constants ns_{32} and ns_{64}. The treatment of sum types is also changed to permit the allocation of tags and tagged records explicit.

5.6.1 Stacks

Stack values in **TALT** are used to represent the actual control stack of a program. Stack slots store return addresses of functions, as well as variables which cannot be spilled into registers. A stack is either empty, or a value pushed onto a stack. Stacks in **TALT** are permitted to contain either 32 or 64-bit values. Pointers into the stack are permitted in a limited fashion. The meta-variable s is used to stand for stack values, and the meta-variable σ is used for the type of stacks. Stack types closely parallel the term level constructs, with the addition of the notion of a compound stack $\sigma_1 \circ \sigma_2$ which stands for prepending all of σ_1 to σ_2.

5.6.2 Values

Small values in **TALT** are syntactically divided into 32 and 63-bit sizes, indicated by the meta-variables w and l respectively. The types of 32-bit values have kind T_{32} and are indicated by the meta-variable τ, whereas the types of 64-bit values have kind T_{64} and are indicated by the meta-variable ϕ. The two forms of 64-bit values are ieee floating point numbers or nonsense. 32-bit values include integers, sum tags, pointers into the stack, nonsense values, and labels.

$$
\begin{array}{lll}
k & ::= & \ldots \mid ST \\
c, \tau, \phi, \sigma & ::= & \ldots \boxed{\texttt{delete Sum}, \texttt{KSum}, \rightarrow} \\
 & & \mid \epsilon \mid \tau_1 \triangleright_{32} c \mid \phi_1 \triangleright_{64} c \mid \sigma_1 \circ \sigma_2 \mid \texttt{sptr}\, \sigma \\
 & & \mid ns_{64} \mid ns_{32} \mid \Gamma \rightarrow 0 \mid \bigvee \\
\Gamma & ::= & \{\mathbf{r}_1{:}\tau_1, \mathbf{r}_2{:}\tau_2, \mathbf{r}_e{:}\tau_e, \mathbf{r}_t{:}\tau_t, \mathbf{f}_1{:}\phi_1, \mathbf{f}_2{:}\phi_2, \mathbf{sp} : \sigma\} \\
\mathbf{r} & ::= & \mathbf{r}_1 \mid \mathbf{r}_2 \mid \mathbf{r}_e \mid \mathbf{r}_t \\
\mathbf{f} & ::= & \mathbf{f}_1 \mid \mathbf{f}_2 \\
q & ::= & \texttt{roll}_c \mid \texttt{unroll}_c \mid \texttt{inj\_union}^{j}_{(i,c)} \mid \texttt{forgetunion} \\
 & & \mid \texttt{pack}[\tau]c \mid \texttt{inj\_dyn}_\tau \\
s & ::= & \epsilon \mid w \triangleright_{32} s \mid l \triangleright_{64} s \\
l & ::= & \mathfrak{r} \mid ns_{64} \\
w & ::= & \ell \mid i \mid ns_{32} \mid \texttt{tag}_i \mid w[\vec{c}] \mid \texttt{sptr}\, i \mid q\ w \\
fv & ::= & l \mid \mathbf{f} \mid \mathbf{sp}(i) \\
sv & ::= & \mathbf{r} \mid \mathbf{sp}(i) \mid w \mid sv[\vec{c}] \mid q\ sv \\
i & ::= & \texttt{mov}\, \mathbf{r}, sv \mid \texttt{loadr}\, \mathbf{r}, \mathbf{r}(i) \mid \texttt{store}\, \mathbf{r}(i), sv \\
 & & \mid \texttt{malloc}\, \mathbf{r}[\tau_1, \ldots, \tau_n]\langle sv_1, \ldots, sv_n\rangle \mid \texttt{malloc}_\tau\, \mathbf{r}(sv_1, sv_2) \\
 & & \mid \texttt{malloc}_\phi\, \mathbf{r}, fv \mid \texttt{malloc}_\phi\, \mathbf{r}(sv, fv) \\
 & & \mid \texttt{call}\, sv \mid \texttt{brtag}_i\, \mathbf{r}, sv \mid \texttt{brtgd}_i\, \mathbf{r}, sv \mid \texttt{dyntag}_c\, \mathbf{r} \\
 & & \mid \texttt{swrite}\, \mathbf{sp}(i), sv \mid \texttt{salloc}\, n \mid \texttt{sfree}\, n \\
 & & \mid \texttt{mov}\, \mathbf{r}, \mathbf{sp} \mid \texttt{mov}\, \mathbf{sp}, sv \\
 & & \mid \texttt{unpack}[\alpha, \mathbf{r}], sv \\
 & & \mid \texttt{sub}_\tau\, \mathbf{r}, sv_1, sv_2 \mid \texttt{upd}_\tau\, sv_1, sv_2, sv_3 \\
 & & \mid \texttt{fmov}\, \mathbf{f}, fv \mid \texttt{floadr}\, \mathbf{f}, \mathbf{r} \mid \texttt{fstore}\, \mathbf{r}, fv \\
 & & \mid \texttt{fswrite}\, \mathbf{sp}(i), fv \\
 & & \mid \texttt{sub}_\phi\, \mathbf{f}, sv_1, sv_2 \mid \texttt{upd}_\phi\, sv_1, sv_2, fv \\
ti & ::= & \texttt{vcase}[\alpha_1.\, \texttt{dead}\, sv, \alpha_2]\, c \\
 & & \mid \texttt{vcase}[\alpha_1, \alpha_2.\, \texttt{dead}\, sv]\, c \\
 & & \mid \texttt{refine}[\langle \beta, \gamma \rangle]\, c \mid \texttt{refine}[\texttt{fold}\, \beta]\, c \\
I & ::= & \texttt{ret} \mid \texttt{jmp}\, sv \mid \texttt{halt}_\tau \mid i; I \mid ti; I \\
hval & ::= & \langle \vec{w} \rangle \mid [\vec{w}] \mid [\vec{l}] \mid l \mid \texttt{dtag} \\
 & & \mid \texttt{code}[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n].\Gamma.I \\
H & ::= & \{\ell_1 \mapsto hval_1, \ldots, \ell_n \mapsto hval_n\} \\
R & ::= & \{\mathbf{r}_1 \mapsto w_1, \mathbf{r}_2 \mapsto w_2, \mathbf{r}_e \mapsto w_e, \mathbf{r}_t \mapsto w_t, \\
 & & \quad \mathbf{f}_1 \mapsto l_1, \mathbf{f}_2 \mapsto l_2{:}\mathbf{sp} \mapsto s\} \\
P & ::= & (H, R, I)
\end{array}
$$

Figure 9: **TALT** syntax

Coercions applied to values ($q\,w$) change the type of the value but have no runtime effect. Labels serve as pointers to values allocated on the heap. As with the version of the **LIL** extended with allocation primitives, a heap type Ψ maps labels to the types of the values to which they point.

5.6.3 Registers and stack slots

Instruction operands are similarly divided into 32 and 64-bit versions. In addition to simple values, operands can be registers or stack slots.

There are three sorts of registers in **TALT**: 32-bit registers, 64-bit registers, and the stack register. By convention, I reserve one of the 32-bit registers for the exception record (see section below). Registers are used to hold the operands and intermediate values, and to pass arguments to functions. The registers given in the version of **TALT** described here are not specialized to the x86 platform that I eventually intend to target.

The stack register is used to maintain the control stack. Reading and writing stack slots is done using offsets from the front of the stack given in 32 bit strides. Allocation of and de-allocation of stack space is handled by special instructions. Newly allocated stack space contains nonsense values which can subsequently be overwritten. Stack slots are used in for the same purposes as the 32 and 64-bit registers, as well as for holding return addresses. The `call` instruction pushes a return address onto the stack, which can subsequently be used by the `ret` instruction.

The type of the register file (and consequently the stack as well) is given by a register file type Γ which maps each register to the type of its contents.

5.6.4 Instructions and instruction sequences

The instruction set described here is very small and focuses mainly on the instructions that are interesting from a typing standpoint. Instructions are provided for moving values between the heap, the stack, and the register file as well as allocating and deallocating space on the stack and the heap. Primitive array instructions are provided to handle bounds checking internally. Instructions such as the array operations and the memory allocation operation are intended to be implemented as assembler macros, since they are not directly provided by the machine.

In addition to standard instructions, there are several type instructions which have no run-time effect. The `vcase` instructions correspond to the `vcase` constructs of the **LIL**, and the `refine` instructions correspond to the path refinement constructs. These instructions can be eliminated when types are removed.

Instruction sequences are a sequence of instructions and type instructions, terminated by either `ret`, $\texttt{jmp}\,sv$, or $\texttt{halt}_\tau$. The return instruction expects the return address to be the top-most element of the stack, and pops it off before returning.

5.6.5 Heap values and heaps

Values that are not guaranteed to fit in 32 or 64-bit slots are always allocated on the heap. Aggregate heap values include tuples of 32-bit values, arrays of 32 or 64-bit values. Individual 64-bit values are also permitted to be allocated on the heap to allow them to be "boxed" and used in contexts expecting 32-bit values. The `dtag` value serves as a place-holder for exception tags, where the label serves as the unique identifier of the tag. Finally, code blocks are also allocated in the heap.

Heaps are simply mappings from labels to heap values. Values in the heap may contain references to other values in the heap, but must be closed with respect to types.

5.7 Translating LIL to TALT

The translating from **LIL** to **TALT** is relatively simple compared to the translation from the **MIL**. For the most part, the **TALT** type system is simply an extension of the **LIL** system. A few notable changes at the type level are discussed in more detail below.

Translating terms from **LIL** to **TALT** mainly serves to make the control structure implicit in switches, exceptions and function calls explicit in the program as jumps and branches. Since **TILT** uses a stack to allocate activation records, it is also necessary to account for this in the translation. Stacks have been dealt with before in the context of typed assembly language, and the translation to **TALT** is not substantially different in nature from the translations previously mapped out by Morrisett, et al. [MWCG98, MCGW98].

5.7.1 Types

The translation of **LIL** types to **TALT** types is relatively straightforward, as there is little change between them. The most notable cases are the translations of sum types and function types. Sum types in the **LIL** are translated into **TALT** by making the types of all of the tags explicit, and function types are translated by making the registers, control stack, and return continuation explicit in the type.

$$|\,\mathtt{Sum}(i)(\tau_0,\ldots,\tau_k)| \stackrel{\text{def}}{=} \bigvee(\mathtt{Tag}_0){::}\ldots{::}(\mathtt{Tag}_{i-1}){::}(\mathtt{Tag}_i \times |\tau_0|){::}\ldots{::}(\mathtt{Tag}_{i+k} \times |\tau_k|)$$

$$\begin{array}{l} |\rightarrow([\tau_0,\ldots,\tau_i])([\phi_0,\ldots,\phi_j])(\tau)| \stackrel{\text{def}}{=} \\ \qquad \forall[\rho_1{:}ST,\rho_2{:}ST]. \\ \qquad\quad \{\mathbf{r}_1{:}|\tau_0|,\mathbf{r}_2{:}|\tau_1|,\mathbf{f}_1{:}|\phi_0|,\mathbf{f}_2{:}|\phi_1|, \\ \qquad\quad \mathbf{r}_e{:}\,\mathbf{Exnpntr}(\rho_2),\mathbf{r_t}{:}ns_{32}, \\ \qquad\quad \mathbf{sp}{:}(\tau_{\mathbf{cont}}|\tau|) \rhd_{32} |\tau_2| \rhd_{32} \ldots \rhd_{32} |\tau_i| \rhd_{32} |\phi_2| \rhd_{64} \ldots \rhd_{64} |\phi_k| \rhd_{64} \rho_1 \circ \rho_2\} \rightarrow 0 \end{array}$$

Functions expect their first two arguments in registers, and all others on the stack. The top most item on the stack upon entry to the function is the address of the code to which the function should return. The type of this code is given by

$$\begin{array}{l} \tau_{\mathbf{cont}} \stackrel{\text{def}}{=} \lambda(\alpha_{\mathbf{ret}}{:}\mathrm{T}_{32}). \\ \qquad \{\mathbf{r}_1{:}ns_{32},\mathbf{r}_2{:}ns_{32},\mathbf{f}_1{:}ns_{64},\mathbf{f}_2{:}ns_{64},\mathbf{r}_e{:}\,\mathbf{Exnpntr}(\rho_2),\mathbf{r_t}{:}\alpha_{\mathbf{ret}}, \\ \qquad \mathbf{sp}{:}ns_{32} \rhd_{32} \ldots \rhd_{32} ns_{32} \rhd_{32} ns_{64} \rhd_{64} \ldots \rhd_{64} ns_{64} \rhd_{64} \rho_1 \circ \rho_2\} \rightarrow 0 \end{array}$$

The return value is passed in $\mathbf{r_t}$ by convention. The return continuation cannot assume anything about the argument slots, which must be coerced to a nonsense value before returning.

The treatment of exceptions in the translation is different from that used by the stack tal paper in which handlers are stored on the stack. To correspond more closely with the current **TILT** implementation, I reserve a single register to hold the current exception record. Exception records contain the address of the current handler, a pointer to a record containing the spilled registers of the handler, and a pointer to the handler's stack. The previous exception record is stored in with the rest of the spilled registers.

The type given to the exception register in function types reflects this. The exception register contains a pointer to an exception record which holds the current handler, a pointer to its stack, and a record of its free variables. The type of the record of free variables is existentially quantified in much the same fashion as a closure.

$$\begin{array}{lll} \mathbf{Exnptr} & \stackrel{\text{def}}{=} & \lambda(\alpha{:}ST).\exists(\beta{:}T).\times[\mathbf{Exnhndler}(\alpha)(\beta),\alpha,\beta] \\ \mathbf{Exnhndler} & \stackrel{\text{def}}{=} & \lambda(\alpha{:}ST).\lambda(\beta{:}\mathrm{T}_{32}). \\ & & \quad \forall[].\{\mathbf{r}_1{:}\mathrm{Exn},\mathbf{r}_2{:}\beta,\mathbf{r}_t{:}ns_{32},\mathbf{r}_e{:}ns_{32},\mathbf{sp}{:}\alpha,\mathbf{f}_1{:}ns_{64},\mathbf{f}_1{:}ns_{64}\} \rightarrow 0 \end{array}$$

The type of the exception handler itself is parameterized both by the type of the handler stack, and by the type of the environment containing its free variables. Notice that the handler expects an exception packet in $\mathbf{r}_1$ and the record of its free variables in $\mathbf{r}_2$.

References

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In Thomas Reps, editor, *Conference Record of the 27th Annual Symposium on Principles of Programming Languages (POPL'00)*, pages 243–253, Boston, Massachusetts, January 2000. ACM Press.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*. IEEE Computer Society Press, June 2001.

[CLN+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00)*, pages 95–107, Vancouver, Canada, June 2000. ACM Press.

[Cra00] Karl Crary. Sound and complete elimination of singleton kinds. Technical Report CMU-CS-00-104, School of Computer Science, Carnegie Mellon University, 2000.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, Paris, France, September 1999. ACM Press.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998. Extended version published as Cornell University technical report TR98-1721.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 262–275, San Antonio, TX, January 1999. ACM Press.

[GM99] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 262–275. ACM Press, January 1999.

[GM00] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *2000 ACM SIGPLAN Workshop on Types in Compilation '00*, Montreal, Canada, September 2000.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, Georgia, May 1999.

[MCGW98] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second Workshop on Types in Compilation*, March 1998.

[MH98] Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In Andrew Gordon, Andrew Pitts, and Carolyn Talcott, editors, *Second Workshop on Higher-Order Techniques in Operational Semantics (HOOTS-II)*, volume 10 of *Electronic Notes in Theoretical Computer Science*, pages ??–??, Stanford University, Stanford, California, December 1998. Elsevier. URL: `http://www.elsevier.nl/locate/entcs/volume10.html`.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[MWCG97] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. Technical Report TR97-1651, Department of Computer Science, Cornell University, 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[Nec97] George Necula. Proof-carrying code. In N. D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97*, pages 106–119, Paris, France, 1997. ACM Press.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

[PCHS00] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, School of Computer Science, Carnegie Mellon University, December 2000.

[SH99] Christopher A. Stone and Robert Harper. Deciding Type Equivalence in a Language with Singleton Kinds. Technical Report CMU-CS-99-155, Department of Computer Science, Carnegie Mellon University, 1999.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[XH99] Hongwei Xi and Robert Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Computer Science Department, Oregon Graduate Institute, July 1999.

A LIL Static semantics

Definitions

$\square$

$$\kappa\,\texttt{list} \stackrel{\text{def}}{=} \mu j.1 + \kappa \times j$$

$$\texttt{nat} \stackrel{\text{def}}{=} \mu j.1 + j$$

$$\begin{array}{ll}\overline{0} & \stackrel{\text{def}}{=} \texttt{fold}_{\texttt{nat}}\,\texttt{inj}_1^{1+\texttt{nat}} *\\ \overline{n+1} & \stackrel{\text{def}}{=} \texttt{fold}_{\texttt{nat}}\,\texttt{inj}_2^{1+\texttt{nat}}(\overline{n})\end{array}$$

Well-formed Kind $\boxed{\Delta \vdash \kappa}$

$$\frac{}{\Delta \vdash \mathrm{T}_{32}} \qquad \frac{}{\Delta \vdash \mathrm{T}_{64}} \qquad \frac{}{\Delta \vdash 1} \qquad \frac{}{\Delta, j, \Delta' \vdash j}$$

$$\frac{\Delta \vdash \kappa_1 \quad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 \to \kappa_2} \qquad \frac{\Delta \vdash \kappa_1 \quad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 \times \kappa_2} \qquad \frac{\Delta \vdash \kappa_1 \quad \Delta \vdash \kappa_2}{\Delta \vdash \kappa_1 + \kappa_2}$$

$$\frac{\Delta, j \vdash \kappa}{\Delta \vdash \mu j.\kappa}\ (j \notin \Delta, j \text{ only positive in } \kappa) \qquad \frac{\Delta, j \vdash \kappa}{\Delta \vdash \forall j.\kappa}\ (j \notin \Delta)$$

Well-formed Constructor $\boxed{\Delta \vdash c : \kappa}$

Constants and their kinds
$\texttt{Float}{:}\mathrm{T}_{64}$ $\quad$ $\texttt{Int}{:}\mathrm{T}_{32}$ $\quad$ $\texttt{Void}{:}\mathrm{T}_{32}$
$\texttt{Array}_{32}{:}\mathrm{T}_{32} \to \mathrm{T}_{32}$ $\quad$ $\texttt{Array}_{64}{:}\mathrm{T}_{64} \to \mathrm{T}_{32}$ $\quad$ $\texttt{Boxed}{:}\mathrm{T}_{64} \to \mathrm{T}_{32}$
$\texttt{Tag}{:}\texttt{nat} \to \mathrm{T}_{32}$ $\quad$ $\texttt{Dyntag}{:}\mathrm{T}_{32} \to \mathrm{T}_{32}$ $\quad$ $\texttt{Dyn}{:}\mathrm{T}_{32}$
$\times{:}\mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{32}$ $\quad$ $\to{:}\mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{64}\texttt{list} \to \mathrm{T}_{32} \to \mathrm{T}_{32}$
$\texttt{Sum}{:}\texttt{nat} \to \mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{32}$ $\quad$ $\texttt{KSum}{:}\texttt{nat} \to \texttt{nat} \to \mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{32}$
$\forall{:}\forall j.(j \to \mathrm{T}_{32}) \to \mathrm{T}_{32}$ $\quad$ $\exists{:}\forall j.(j \to \mathrm{T}_{32}) \to \mathrm{T}_{32}$
$\texttt{Rec}{:}\forall j.((j \to \mathrm{T}_{32}) \to (j \to \mathrm{T}_{32})) \to j \to \mathrm{T}_{32}$

$$\frac{}{\Delta \vdash * : 1} \qquad \frac{}{\Delta, \alpha{:}\kappa, \Delta' \vdash \alpha : \kappa}$$

$$\frac{\Delta \vdash \kappa_1 \quad \Delta, \alpha{:}\kappa_1 \vdash c : \kappa_2}{\Delta \vdash \lambda(\alpha :: \kappa).c : \kappa_1 \to \kappa_2}\ \alpha \notin \Delta \qquad \frac{\Delta \vdash c_1 : \kappa_1 \to \kappa_2 \quad \Delta \vdash c_2 : \kappa_1}{\Delta \vdash c_1\, c_2 : \kappa_2}$$

$$\frac{\Delta \vdash c_1 : \kappa_1 \quad \Delta \vdash c_2 : \kappa_2}{\langle c_1, c_2 \rangle \vdash \kappa_1 \times \kappa_2 :} \qquad \frac{\Delta \vdash c : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_i\, c : \kappa_i}\ (i \in \{1,2\})$$

$$\frac{\Delta \vdash c : \kappa_i \quad (i \in \{1,2\})}{\Delta \vdash \texttt{inj}_i^{\kappa_1+\kappa_2}\, c : \kappa_1 + \kappa_2} \qquad \frac{\begin{array}{c}\Delta \vdash c : \kappa_1 + \kappa_2 \\ \Delta, \alpha_1{:}\kappa_1 \vdash c_1 : \kappa \\ \Delta, \alpha_2{:}\kappa_2 \vdash c_2 : \kappa\end{array}}{\Delta \vdash \texttt{case}\, c(\alpha_1.c_1, \alpha_2.c_2) : \kappa}\ \alpha_1, \alpha_2 \notin \Delta$$

$$\frac{\Delta \vdash \mu j.\kappa \quad \Delta \vdash c : \kappa[\mu j.\kappa / j]}{\Delta \vdash \texttt{fold}_{\mu j.\kappa}\, c : \mu j.\kappa}$$

$$\frac{\begin{array}{c}\Delta \vdash \mu j.\kappa \quad \Delta \vdash \kappa' \quad j, \alpha, \rho, \iota \notin \Delta \\ \Delta, j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}(j \to \mu j.\kappa) \vdash c : \kappa'\end{array}}{\Delta \vdash \texttt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}(j \to \mu j.\kappa)\ \texttt{in}\, c) : \mu j.\kappa \to \kappa'}$$

$$\frac{\Delta \vdash c : \forall j.\kappa' \quad \Delta \vdash \kappa}{\Delta \vdash c[\kappa] : \kappa'[\kappa / j]} \qquad \frac{\Delta, j \vdash c : \kappa}{\Delta \vdash \Lambda j.c : \forall j.\kappa}\ j \notin \Delta$$

Constructor Equivalence $\boxed{\Delta \vdash c \equiv c' : \kappa}$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c \equiv c : \kappa} \qquad \frac{\Delta \vdash c' \equiv c : \kappa}{\Delta \vdash c \equiv c' : \kappa} \qquad \frac{\Delta \vdash c_1 \equiv c_2 : \kappa \quad \Delta \vdash c_2 \equiv c_3 : \kappa}{\Delta \vdash c_1 \equiv c_3 : \kappa}$$

$$\frac{\Delta \vdash c : \kappa_1 \to \kappa_2 \quad \alpha \notin \mathrm{FV}(c)}{\Delta \vdash \lambda(\alpha{:}\kappa_1).c\alpha \equiv c : \kappa_1 \to \kappa_2} \qquad \frac{\Delta \vdash c : \kappa_1 \times \kappa_2}{\Delta \vdash \langle \pi_1\, c, \pi_2\, c\rangle \equiv c : \kappa_1 \times \kappa_2}$$

$$\frac{\Delta \vdash c : \kappa_1 + \kappa_2}{\Delta \vdash \mathtt{case}(c, \alpha.\, \mathtt{inj}_1^{\kappa_1+\kappa_2}\, \alpha, \alpha.\, \mathtt{inj}_2^{\kappa_1+\kappa_2}\, \alpha) \equiv c : \kappa_1 + \kappa_2}$$

$$\frac{\Delta \vdash c_1 : \kappa_1 \quad \Delta \vdash c_2 : \kappa_2}{\Delta \vdash \pi_1 \langle c_1, c_2\rangle \equiv c_1 : \kappa_1} \qquad \frac{\Delta \vdash c_1 : \kappa_1 \quad \Delta \vdash c_2 : \kappa_2}{\Delta \vdash \pi_2 \langle c_1, c_2\rangle \equiv c_2 : \kappa_2}$$

$$\frac{\begin{array}{c}\Delta \vdash \kappa_1 \\ \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa_2 \quad \Delta \vdash c_2 : \kappa_1\end{array}}{\Delta \vdash (\lambda(\alpha{:}\kappa_1).c_1)c_2 \equiv c_1[c_2/\alpha] : \kappa_2} \qquad \frac{\Delta \vdash \kappa \quad \Delta, j \vdash c : \kappa'}{\Delta \vdash (\Lambda j.c)\kappa \equiv c[\kappa/j] : \kappa'[\kappa/j]}$$

$$\frac{\begin{array}{c}\Delta \vdash c : \kappa_1 \quad \Delta \vdash \kappa_2 \\ \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa \quad \Delta, \alpha{:}\kappa_2 \vdash c_2 : \kappa\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_1^{\kappa_1+\kappa_2}\, c, \alpha.c_1, \alpha.c_2) \equiv c_1[c/\alpha] : \kappa}$$

$$\frac{\begin{array}{c}\Delta \vdash c : \kappa_2 \quad \Delta \vdash \kappa_1 \\ \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa \quad \Delta, \alpha{:}\kappa_2 \vdash c_2 : \kappa\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_2^{\kappa_1+\kappa_2}\, c, \alpha.c_1, \alpha.c_2) \equiv c_2[c/\alpha] : \kappa}$$

$$\frac{\begin{array}{c}\Delta \vdash c' : \kappa[\mu j.\kappa/j] \quad \Delta \vdash \kappa' \quad (j, \alpha, \rho, \iota \notin \Delta) \\ \Delta, j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}(j \to \mu j.\kappa) \vdash c : \kappa' \quad \Delta \vdash \mu j.\kappa\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}(j \to \mu j.\kappa)\ \mathtt{in}\, c)\ \mathtt{fold}_{\mu j.\kappa}\, c' \\ \qquad \equiv c[\mu j.\kappa, c', \mathtt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}(j \to \mu j.\kappa)\ \mathtt{in}\, c), \lambda(\alpha{:}\mu j.\kappa).\alpha/j, \alpha, \rho, \iota] \\ \hfill : \mu j.\kappa \to \kappa'\end{array}}$$

$$\frac{\begin{array}{c}\Delta \vdash \kappa_1 \\ \Delta, \alpha{:}\kappa_1 \vdash c_1 \equiv c_2 : \kappa_2\end{array}}{\Delta \vdash \lambda(\alpha{:}\kappa_1).c_1 \equiv \lambda(\alpha{:}\kappa_1).c_2 : \kappa_1 \to \kappa_2} \qquad \frac{\begin{array}{c}\Delta \vdash c_1 \equiv c_1' : \kappa_1 \to \kappa_2 \\ \Delta \vdash c_2 \equiv c_2' : \kappa_1\end{array}}{\Delta \vdash c_1 c_2 \equiv c_1' c_2' : \kappa_2}$$

$$\frac{\Delta \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_1\, c \equiv \pi_1\, c' : \kappa_1} \qquad \frac{\Delta \vdash c \equiv c' : \kappa_2 \quad \Delta \vdash \kappa_1 \quad \Delta \vdash \kappa_2}{\Delta \vdash \mathtt{inj}_2^{\kappa_1+\kappa_2}\, c \equiv \mathtt{inj}_2^{\kappa_1+\kappa_2}\, c' : \kappa_1 + \kappa_2}$$

$$\frac{\Delta \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_2\, c \equiv \pi_2\, c' : \kappa_2} \qquad \frac{\Delta \vdash c \equiv c' : \kappa_1 \quad \Delta \vdash \kappa_1 \quad \Delta \vdash \kappa_2}{\Delta \vdash \mathtt{inj}_1^{\kappa_1+\kappa_2}\, c \equiv \mathtt{inj}_1^{\kappa_1+\kappa_2}\, c' : \kappa_1 + \kappa_2}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv c' : \kappa_1 + \kappa_2 \\ \Delta, \alpha{:}\kappa \vdash c_1 \equiv c_1' : \kappa \quad \Delta, \alpha{:}\kappa \vdash c_2 \equiv c_2' : \kappa\end{array}}{\Delta \vdash \mathtt{case}(c, \alpha.c_1, \alpha.c_2) \equiv \mathtt{case}(c', \alpha.c_1', \alpha.c_2') : \kappa}$$

$$\frac{\Delta \vdash c \equiv c' : \kappa[\mu j.\kappa/j]}{\Delta \vdash \mathtt{fold}_{\mu j.\kappa}\, c \equiv \mathtt{fold}_{\mu j.\kappa}\, c' : \mu j.\kappa}$$

$$\frac{\begin{array}{c}j, \alpha, \rho, \iota \notin \Delta \\ \Delta \vdash \mu j.\kappa \quad \Delta \vdash \kappa' \quad \Delta, j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}(j \to \mu j.\kappa) \vdash c \equiv c' : \kappa'\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}\kappa, \rho{:}j \to \kappa', \iota{:}j \to \mu j.\kappa \,\mathrm{in}\, c) \\ \quad \equiv \mathtt{pr}(j, \alpha{:}\kappa, \rho{:}j \to \kappa', \iota{:}j \to \mu j.\kappa \,\mathrm{in}\, c') : \mu j.\kappa \to \kappa'\end{array}}$$

$$\frac{\Delta, j \vdash c \equiv c' : \kappa'}{\Delta \vdash \Lambda j.c \equiv \Lambda j.c' : \forall j.\kappa'}\; j \notin \Delta \qquad \frac{\Delta \vdash c \equiv c' : \forall j.\kappa' \quad \Delta \vdash \kappa}{\Delta \vdash c[\kappa] \equiv c'[\kappa] : \kappa'[\kappa/j]}$$

Well-formed 64 bit value $\boxed{\Delta; \Gamma \vdash fv : \phi}$

$$\frac{}{\Delta; \Gamma \vdash \mathfrak{r} : \mathtt{Float}} \qquad \frac{}{\Delta; \Gamma, x_{64}{:}\phi, \Gamma' \vdash x_{64} : \phi}$$

Well-formed 32 bit Value $\boxed{\Delta; \Gamma \vdash sv : \tau}$

$$\frac{}{\Delta; \Gamma, x{:}\tau, \Gamma' \vdash x : \tau} \qquad \frac{}{\Delta; \Gamma \vdash i : \mathtt{Int}}$$

$$\frac{\Delta; \Gamma \vdash fv : \phi}{\Delta; \Gamma \vdash \mathtt{box}\, fv : \mathtt{Boxed}\, \phi} \qquad \frac{\Delta; \Gamma \vdash sv_i : \tau_i \quad i \in 0, \ldots, n}{\Delta; \Gamma \vdash \langle sv_0, \ldots, sv_n \rangle : \times [\tau_0, \ldots, \tau_n]}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash sv : \mathtt{Tag}\, \overline{k} \\ \Delta \vdash c : \mathrm{T}_{32}\ \mathtt{list} \quad k < i\end{array}}{\Delta; \Gamma \vdash \mathtt{inj\_tag}^k_{(i,c)}\, sv : \mathtt{KSum}(\overline{k})(\bar{i})(c)} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash sv : \times [\mathtt{Tag}(\overline{k}), c_k] \\ \Delta \vdash c \equiv c_i{::}\ldots{::}c_k{::}c' : \mathrm{T}_{32}\mathtt{list}\end{array}}{\Delta; \Gamma \vdash \mathtt{inj}^k_{(i,c)}\, sv : \mathtt{KSum}(\overline{k})(\bar{i})(c)}$$

$$\frac{\Delta; \Gamma \vdash sv_1 : \mathtt{Dyntag}\, \tau \quad \Delta; \Gamma \vdash sv_2 : \tau}{\Delta; \Gamma \vdash \mathtt{inj\_dyn}_\tau(sv_1, sv_2) : \mathtt{Dyn}} \qquad \frac{\begin{array}{c}\Delta \vdash \tau \equiv \mathtt{Rec}[\kappa](c)(c_p) : \mathrm{T}_{32} \\ \Delta; \Gamma \vdash sv : c(\mathtt{Rec}[\kappa]c)c_p\end{array}}{\Delta; \Gamma \vdash \mathtt{roll}_\tau\, sv : \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau \equiv \exists[\kappa](c') : \mathrm{T}_{32} \quad \Delta \vdash c : \kappa \\ \Delta; \Gamma \vdash sv : c'c]\end{array}}{\Delta; \Gamma \vdash \mathtt{pack}\, sv\, \mathtt{as}\, \tau\, \mathtt{hiding}\, c : \tau} \qquad \frac{\Delta; \Gamma \vdash sv : \forall[\kappa](c') \quad \Delta \vdash c : \kappa}{\Delta; \Gamma \vdash sv[c] : c'c}$$

$$\frac{\Delta; \Gamma \vdash sv : \mathtt{KSum}(k)(i)(c)}{\Delta; \Gamma \vdash \mathtt{forgetknown}\, sv : \mathtt{Sum}(i)(c)} \qquad \frac{}{\Delta; \Gamma \vdash \mathtt{tag}_i : \mathtt{Tag}(\bar{i})}$$

Well formed 64 bit operations $\boxed{\Delta;\Gamma \vdash \mathit{fopr} : \phi\ \ \mathbf{opr}_{64}}$

$$\frac{\Delta;\Gamma \vdash \mathit{fv} : \phi}{\Delta;\Gamma \vdash \mathit{fv} : \phi\ \ \mathbf{opr}_{64}} \qquad \frac{\Delta;\Gamma \vdash sv : \mathtt{Boxed}\,\phi}{\Delta;\Gamma \vdash \mathtt{unbox}\,sv : \phi\ \ \mathbf{opr}_{64}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv_1 : \mathtt{Array}_{64}\phi \\ \Delta;\Gamma \vdash sv_2 : \mathtt{Int}\end{array}}{\Delta;\Gamma \vdash \mathtt{sub}_\phi(sv_1, sv_2) : \phi\ \ \mathbf{opr}_{64}}$$

Well-formed 32 bit operation $\boxed{\Delta;\Gamma \vdash \mathit{opr} : \tau\ \ \mathbf{opr}_{32}}$

$$\frac{\Delta;\Gamma \vdash sv : \tau}{\Delta;\Gamma \vdash sv : \tau\ \ \mathbf{opr}_{32}} \qquad \frac{\begin{array}{c}\Delta \vdash \tau \equiv \mathtt{Rec}[\kappa](c)(c_p) : \mathrm{T}_{32} \\ \Delta;\Gamma \vdash sv : \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{unroll}_\tau\, sv : c(\mathtt{Rec}[\kappa]c)c_p\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta;\Gamma \vdash sv : \times(\tau_0 :: \ldots :: \tau_i :: c')}{\Delta;\Gamma \vdash \mathtt{select}^i\, sv : \tau_i\ \ \mathbf{opr}_{32}} \qquad \frac{\Delta;\Gamma \vdash sv : \mathtt{KSum}(\overline{k})(\bar{i})(\tau_i :: \ldots :: \tau_k :: c')}{\Delta;\Gamma \vdash \mathtt{proj}^k\, sv : \times[\mathtt{Tag}(\overline{k}), \tau_k]\ \ \mathbf{opr}_{32}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv : \mathtt{Sum}(i)([\tau_{\hat{i}}, \ldots, \tau_k]) \\ \Delta;\Gamma, x_k{:}\,\mathtt{KSum}(\overline{k})(i)([\tau_{\hat{i}}, \ldots, \tau_k]) \vdash e_k : \tau\ \ \mathbf{exp}\end{array}}{\Delta;\Gamma \vdash \mathtt{case}(sv)(x.e_1, \ldots, x.e_k) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta \vdash \tau : \mathrm{T}_{32}}{\Delta;\Gamma \vdash \mathtt{dyntag}_\tau : \mathtt{Dyntag}\,\tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv : \mathtt{Dyn} \quad \Delta;\Gamma \vdash sv_1 : \mathtt{Dyntag}\,\tau_1 \\ \Delta;\Gamma, x_1{:}\tau_1 \vdash e_1 : \tau\ \ \mathbf{exp} \quad \Delta;\Gamma \vdash e : \tau\ \ \mathbf{exp}\end{array}}{\Delta;\Gamma \vdash \mathtt{dyncase}(sv)(sv_1 \Rightarrow x_1.e_1, \_ \Rightarrow e) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta;\Gamma \vdash sv : \mathtt{Dyn} \quad \Delta \vdash \tau : \mathrm{T}_{32}}{\Delta;\Gamma \vdash \mathtt{raise}_\tau\, sv : \tau\ \ \mathbf{opr}_{32}} \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \tau\ \ \mathbf{exp} \\ \Delta;\Gamma, x{:}\,\mathtt{Dyn} \vdash e_2 : \tau\ \ \mathbf{exp}\end{array}}{\Delta;\Gamma \vdash \mathtt{handle}_\tau(e_1, x.e_2) : \tau\ \ \mathbf{opr}_{32}}\ x \notin \Gamma$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv : \to([\tau_0, \ldots, \tau_n])([\phi_0, \ldots, \phi_k])(\tau) \\ \Delta;\Gamma \vdash sv_i : \tau_i \quad \Delta;\Gamma \vdash \mathit{fv}_i : \phi_i\end{array}}{\Delta;\Gamma \vdash sv(sv_0, \ldots, sv_n)(\mathit{fv}_0, \ldots, \mathit{fv}_k) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\begin{array}{c}\Delta, \alpha_1{:}\kappa_1, \Delta'; \Gamma[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha] \vdash sv[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha] : \mathtt{Void} \\ \Delta, \alpha_2{:}\kappa_2, \Delta'; \Gamma[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha] \vdash e[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha] : \tau[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha]\ \ \mathbf{exp} \\ \Delta, \alpha{:}\kappa_1 + \kappa_2, \Delta'; \Gamma \vdash c \equiv \alpha : \kappa_1 + \kappa_2 \quad \alpha_1, \alpha_2 \notin \Delta, \Delta'\end{array}}{\Delta, \alpha{:}\kappa_1 + \kappa_2, \Delta'; \Gamma \vdash \mathtt{vcase}_\tau(c, \alpha_1.\,\mathtt{dead}\,sv, \alpha_2.e) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\begin{array}{c}\Delta, \alpha_1{:}\kappa_1, \Delta'; \Gamma[\mathtt{inj}_1^{\kappa_1+\kappa_2}\, \alpha_1/\alpha] \vdash e[\mathtt{inj}_1^{\kappa_1+\kappa_2}\, \alpha_1/\alpha] : \tau[\mathtt{inj}_1^{\kappa_1+\kappa_2}\, \alpha_1/\alpha]\ \ \mathbf{exp} \\ \Delta, \alpha_2{:}\kappa_2, \Delta'; \Gamma[\mathtt{inj}_2^{\kappa_1+\kappa_2}\, \alpha_2/\alpha] \vdash sv[\mathtt{inj}_2^{\kappa_1+\kappa_2}\, \alpha_2/\alpha] : \mathtt{Void} \\ \Delta, \alpha{:}\kappa_1+\kappa_2, \Delta' \vdash c \equiv \alpha : \kappa_1+\kappa_2 \quad \alpha_1, \alpha_2 \notin \Delta, \Delta'\end{array}}{\Delta, \alpha{:}\kappa_1+\kappa_2, \Delta' \vdash \mathtt{vcase}_\tau(c, \alpha_1.e, \alpha_2.\,\mathtt{dead}\, sv) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \mathtt{inj}_2^{\kappa_1+\kappa_2}\, c' : \kappa_1+\kappa_2 \\ \Delta; \Gamma \vdash e[c'/\alpha_2] : \tau\ \ \mathbf{exp}\end{array}}{\Delta; \Gamma \vdash \mathtt{vcase}_\tau(c, \alpha_1.\,\mathtt{dead}\, sv, \alpha_2.e) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \mathtt{inj}_1^{\kappa_1+\kappa_2}\, c' : \kappa_1+\kappa_2 \\ \Delta; \Gamma \vdash e[c'/\alpha_1] : \tau\ \ \mathbf{exp}\end{array}}{\Delta; \Gamma \vdash \mathtt{vcase}_\tau(c, \alpha_1.e, \alpha_2.sv) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta; \Gamma \vdash sv_1 : \mathtt{Int} \quad \Delta; \Gamma \vdash sv_2 : \tau}{\Delta; \Gamma \vdash \mathtt{array}_\tau(sv_1, sv_2) : \mathtt{Array}_{32}(\tau)\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta; \Gamma \vdash sv_1 : \mathtt{Array}_{32}(\tau) \quad \Delta; \Gamma \vdash sv_2 : \mathtt{Int}}{\Delta; \Gamma \vdash \mathtt{sub}_\tau(sv_1, sv_2) : \tau\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta; \Gamma \vdash sv_1 : \mathtt{Array}_{32}(\tau) \quad \Delta; \Gamma \vdash sv_2 : \mathtt{Int} \quad \Delta; \Gamma \vdash sv_3 : \tau}{\Delta; \Gamma \vdash \mathtt{upd}_\tau(sv_1, sv_2, sv_3) : \mathtt{Unit}\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta; \Gamma \vdash sv : \mathtt{Int} \quad \Delta; \Gamma \vdash fv : \phi}{\Delta; \Gamma \vdash \mathtt{array}_\phi(sv, fv) : \mathtt{Array}_{64}\phi\ \ \mathbf{opr}_{32}}$$

$$\frac{\Delta; \Gamma \vdash sv_1 : \mathtt{Array}_{64}\phi \quad \Delta; \Gamma \vdash sv_2 : \mathtt{Int} \quad \Delta; \Gamma \vdash fv : \phi}{\Delta; \Gamma \vdash \mathtt{upd}_\phi(sv_1, sv_2, fv) : \mathtt{Unit}\ \ \mathbf{opr}_{64}}$$

Well-formed Expression $\boxed{\Delta; \Gamma \vdash e : \tau\ \ \mathbf{exp}}$

$$\frac{\begin{array}{c}\Delta \vdash \kappa_i \quad \Delta, \vec{\alpha}{:}\vec{\kappa} \vdash \tau_i : \mathrm{T}_{32} \quad \Delta, \vec{\alpha}{:}\vec{\kappa} \vdash \phi_i : \mathrm{T}_{64} \\ \Delta, \vec{\alpha}{:}\vec{\kappa}; \Gamma, f{:}\forall[\vec{\alpha}{:}\vec{\kappa}](\vec{\tau})(\vec{\phi}) \to \tau, \vec{x}{:}\vec{\tau}, \vec{x_{64}}{:}\vec{\phi} \vdash e : \tau\ \ \mathbf{exp} \\ \Delta; \Gamma, f{:}\forall[\vec{\alpha}{:}\vec{\kappa}](\vec{\tau})(\vec{\phi}) \to \tau \vdash e' : \tau'\ \ \mathbf{exp}\end{array}}{\Delta; \Gamma \vdash \mathtt{let\,rec}_\tau\ f[\vec{\alpha}{:}\vec{\kappa}](\vec{x}{:}\vec{\tau})(\vec{x_{64}}{:}\vec{\phi}).e\ \mathtt{in}\, e' : \tau'\ \ \mathbf{exp}}\ \vec{\alpha}, \vec{x}, \vec{x_{64}}, f \notin \Delta, \Gamma$$

$$\frac{\Delta; \Gamma \vdash sv : \tau}{\Delta; \Gamma \vdash sv : \tau\ \ \mathbf{exp}}$$

$$\frac{\Delta;\Gamma \vdash opr : \tau \;\; \mathbf{opr}_{32} \qquad \Delta;\Gamma, x{:}\tau \vdash e : \tau' \;\; \mathbf{exp}}{\Delta;\Gamma \vdash \mathtt{let}\, x = opr\; \mathtt{in}\, e : \tau' \;\; \mathbf{exp}} \qquad \frac{\Delta;\Gamma \vdash opr : \phi \;\; \mathbf{opr}_{32} \qquad \Delta;\Gamma, x_{64}{:}\phi \vdash e : \tau' \;\; \mathbf{exp}}{\Delta;\Gamma \vdash \mathtt{let}\, x_{64} = fopr\; \mathtt{in}\, e : \tau' \;\; \mathbf{exp}}$$

$$\frac{\Delta;\Gamma \vdash sv : \exists[\kappa](c) \qquad \Delta, \alpha{:}\kappa; \Gamma, x{:}(c\alpha) \vdash e : \tau' \;\; \mathbf{exp} \quad \alpha \notin fv(\tau')}{\Delta;\Gamma \vdash \mathtt{let}[\alpha, x] = \mathtt{unpack}\, sv\; \mathtt{in}\, e : \tau' \;\; \mathbf{exp}}\; \alpha, x \notin \Delta, \Gamma$$

$$\frac{\Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2, \Delta'; \Gamma[\langle\beta,\gamma\rangle/\alpha] \vdash e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \;\; \mathbf{exp} \qquad \Delta, \alpha{:}\kappa_1 \times \kappa_2, \Delta' \vdash c \equiv \alpha : \kappa_1 \times \kappa_2}{\Delta, \alpha{:}\kappa_1 \times \kappa_2, \Delta'; \Gamma \vdash \mathtt{let}\langle\beta,\gamma\rangle = c\; \mathtt{in}\, e : \tau \;\; \mathbf{exp}}\; \beta, \gamma \notin \Delta$$

$$\frac{\Delta;\Gamma \vdash e[c_1, c_2/\beta, \gamma] : \tau \;\; \mathbf{exp} \qquad \Delta \vdash c \equiv \langle c_1, c_2\rangle : \kappa_1 \times \kappa_2}{\Delta;\Gamma \vdash \mathtt{let}\langle\beta,\gamma\rangle = c\; \mathtt{in}\, e : \tau \;\; \mathbf{exp}}\; \beta \notin \Delta$$

$$\frac{\Delta, \beta{:}\kappa[\mu j.\kappa/j], \Delta'; \Gamma[\mathtt{fold}_{\mu j.\kappa}\, \beta/\alpha] \vdash e[\mathtt{fold}_{\mu j.\kappa}\, \beta/\alpha] : \tau[\mathtt{fold}_{\mu j.\kappa}\, \beta/\alpha] \qquad \Delta, \alpha{:}\mu j.\kappa, \Delta' \vdash c \equiv \alpha : \mu j.\kappa}{\Delta, \alpha{:}\mu j.\kappa, \Delta'; \Gamma \vdash \mathtt{let}(\mathtt{fold}\, \beta) = c\; \mathtt{in}\, e : \tau \;\; \mathbf{exp}}$$

$$\frac{\Delta;\Gamma \vdash e[c'/\beta] : \tau \;\; \mathbf{exp} \qquad \Delta \vdash c \equiv \mathtt{fold}_{\mu j.\kappa}\, c' : \mu j.\kappa}{\Delta;\Gamma \vdash \mathtt{let}(\mathtt{fold}\, \beta) = c\; \mathtt{in}\, e : \tau \;\; \mathbf{exp}}$$

Well-formed Context $\boxed{\vdash \Delta, \;\; \Delta \vdash \Gamma}$

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Delta}{\vdash \Delta, j}\; j \notin \Delta \qquad \frac{\vdash \Delta \quad \Delta \vdash \kappa}{\vdash \Delta, \alpha{:}\kappa}\; \alpha \notin \Delta$$

$$\frac{\vdash \Delta}{\Delta \vdash \bullet} \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \tau : \mathrm{T}_{32}}{\Delta \vdash \Gamma, x{:}\tau}\; x \notin \Gamma \qquad \frac{\Delta \vdash \Gamma \quad \Delta \vdash \phi : \mathrm{T}_{64}}{\Delta \vdash \Gamma, x_{64}{:}\phi}\; x_{64} \notin \Gamma$$

A.1 New rules for closure converted LIL

Well-formed *hval* $\boxed{\Gamma \vdash h : \tau \;\; \mathbf{hval}}$

$$\frac{\bullet \vdash \kappa_i \quad \vec{\alpha}{:}\vec{\kappa} \vdash \tau_i : \mathrm{T}_{32} \quad \vec{\alpha}{:}\vec{\kappa} \vdash \phi_i : \mathrm{T}_{64} \qquad \vec{\alpha}{:}\vec{\kappa}; \Gamma, \vec{x}{:}\vec{\tau}, \vec{x_{64}}{:}\phi \vdash e : \tau}{\Gamma \vdash \mathtt{code}_{\tau}[\vec{\alpha}{:}\vec{\kappa}](\vec{x}{:}\vec{\tau})(\vec{x_{64}}{:}\vec{\phi}).e : \forall[\vec{\alpha}{:}\vec{\kappa}](\vec{\tau})(\vec{\phi}) \to \tau \;\; \mathbf{hval}}$$

Well-formed program $\boxed{\vdash p : \tau}$

$$\frac{f_1{:}\tau_1, \ldots, f_n{:}\tau_n \vdash hval_i : \tau_i \ \ \textbf{hval} \quad i = 1 \ldots n \qquad \bullet; f_1{:}\tau_1, \ldots, f_n{:}\tau_n \vdash e : \tau}{\vdash \texttt{letrec}\, f_1{:}\tau_1 \mapsto hval_1, \ldots, f_n{:}\tau_n \mapsto hval_n \,\texttt{in}\, e : \tau}$$

A.2 New rules for LIL with memory allocation

New well-formed value rules $\boxed{\Psi; \Delta; \Gamma \vdash sv : \tau}$

$$\frac{}{\Psi_1, \ell{:}\tau, \Psi_2; \Delta; \Gamma \vdash \ell : \tau}$$

New well-formed *opr* rules $\boxed{\Psi; \Delta; \Gamma \vdash opr : \tau}$

$$\frac{\Psi; \Delta; \Gamma \vdash sv_i : \tau_i \quad i = 1 \ldots n}{\Psi; \Delta; \Gamma \vdash \texttt{malloc}[\tau_1, \ldots, \tau_n]\langle sv_1, \ldots, sv_n \rangle : \times[\tau_1, \ldots, \tau_n]}$$

$$\frac{\Psi; \Delta; \Gamma \vdash sv_1 : \texttt{Int} \quad \Psi; \Delta; \Gamma \vdash sv_2 : \tau}{\Psi; \Delta; \Gamma \vdash \texttt{malloc}_\tau(sv_1, sv_2) : \texttt{Array}_{32}(\tau)}$$

$$\frac{\Psi; \Delta; \Gamma \vdash fv : \texttt{Float}}{\Psi; \Delta; \Gamma \vdash \texttt{box}\, fv : \texttt{Boxed}\,\texttt{Float}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash sv_1 : \texttt{Int} \quad \Psi; \Delta; \Gamma \vdash fv : \texttt{Float}}{\Psi; \Delta; \Gamma \vdash \texttt{malloc}_\phi(sv, fv) : \texttt{Array}_{64}\texttt{Float}}$$

Well-formed *hval* $\boxed{\Delta; \Gamma \vdash h : \tau \ \ \textbf{hval}}$

$$\frac{\Psi; \bullet; \bullet \vdash sv_i : \tau_i \quad i = 1 \ldots n}{\Psi \vdash \langle sv_1, \ldots, sv_n \rangle : \times[\tau_1, \ldots, \tau_n] \ \ \textbf{hval}}$$

$$\frac{\Psi; \bullet; \bullet \vdash sv_i : \tau \quad i = 1 \ldots n}{\Psi \vdash [sv_1, \ldots, sv_n] : \texttt{Array}_{32}\tau \ \ \textbf{hval}}$$

$$\frac{\Psi; \bullet; \bullet \vdash fv : \texttt{Float}}{\Psi \vdash fv : \texttt{Boxed}\,\texttt{Float} \ \ \textbf{hval}} \qquad \frac{\Psi; \bullet; \bullet \vdash fv_i : \phi \quad i = 1 \ldots n}{\Psi \vdash [fv_1, \ldots, fv_n] : \texttt{Array}_{64}\phi \ \ \textbf{hval}}$$

$$\frac{}{\Psi \vdash \texttt{exn\_tag} : \texttt{Dyntag}_c \ \ \textbf{hval}}$$

$$\frac{\bullet \vdash \kappa_i \quad \overrightarrow{\alpha{:}\kappa} \vdash \tau_i : \mathrm{T}_{32} \quad \overrightarrow{\alpha{:}\kappa} \vdash \phi_i : \mathrm{T}_{64} \qquad \Psi; \overrightarrow{\alpha{:}\kappa}; \Gamma, \overrightarrow{x{:}\tau}, \overrightarrow{x_{64}{:}\phi} \vdash e : \tau}{\Psi \vdash \texttt{code}_\tau[\overrightarrow{\alpha{:}\kappa}](\overrightarrow{x{:}\tau})(\overrightarrow{x_{64}{:}\phi}).e : \forall[\overrightarrow{\alpha{:}\kappa}](\vec{\tau})(\vec{\phi}) \to \tau \ \ \textbf{hval}}$$

Well-formed program $\boxed{\vdash p : \tau}$

$$\frac{f_1{:}\tau_1, \ldots, f_n{:}\tau_n \vdash hval_i : \tau_i \ \ \textbf{hval} \quad i = 1 \ldots n \qquad f_1{:}\tau_1, \ldots, f_n{:}\tau_n; \bullet; \bullet \vdash e : \tau}{\vdash \texttt{letrec}\, \ell_1{:}\tau_1 \mapsto hval_1, \ldots, \ell_n{:}\tau_n \mapsto hval_n \,\texttt{in}\, e : \tau}$$

B MIL to LIL translation

Definitions for the constructor and kind translations $\square$

$$\mathrm{T}_{\mathrm{mil}} \stackrel{\mathrm{def}}{=} \mathrm{T}_{32}\ \texttt{list} + 1 + \mathrm{T}_{32}$$

$$\begin{array}{l} \mathrm{interp} : \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32} \stackrel{\mathrm{def}}{=} \lambda(\alpha{:}\mathrm{T}_{\mathrm{mil}}). \\ \qquad \texttt{case}\, \alpha \\ \qquad\quad \textbf{Record}\, l \Rightarrow \times(l) \\ \qquad\quad |\, \textbf{BFloat} \Rightarrow \texttt{Boxed}(\texttt{Float}) \\ \qquad\quad |\, \textbf{Other}\, t \Rightarrow t \end{array}$$

$$\begin{array}{lll} \textbf{Other} & : \mathrm{T}_{32} \to \mathrm{T}_{\mathrm{mil}} & \stackrel{\mathrm{def}}{=} \lambda(\alpha{:}\mathrm{T}_{32}).\, \texttt{inj}_2^{\mathrm{T}_{\mathrm{mil}}}(\texttt{inj}_2\, \alpha) \\ \textbf{BFloat} & : \mathrm{T}_{\mathrm{mil}} & \stackrel{\mathrm{def}}{=} \texttt{inj}_2^{\mathrm{T}_{\mathrm{mil}}}(\texttt{inj}_1 *) \\ \textbf{Record} & : \mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{\mathrm{mil}} & \stackrel{\mathrm{def}}{=} \lambda(\alpha{:}\mathrm{T}_{32}\texttt{list}).\, \texttt{inj}_1^{\mathrm{T}_{\mathrm{mil}}}\, \alpha \end{array}$$

Kinds $\boxed{|\kappa|}$

$$\begin{array}{ll} |\mathrm{T}_{32}| & \stackrel{\mathrm{def}}{=} \mathrm{T}_{\mathrm{mil}} \\ |\kappa_1 \to \kappa_2| & \stackrel{\mathrm{def}}{=} |\kappa_1| \to |\kappa_2| \\ |\kappa_1 \times \kappa_2| & \stackrel{\mathrm{def}}{=} |\kappa_1| \times |\kappa_2| \end{array}$$

Types $\boxed{|c|}$

$$
\begin{array}{lll}
|\alpha| & \stackrel{\text{def}}{=} & \alpha \\
|\lambda(\alpha{:}\kappa).c| & \stackrel{\text{def}}{=} & \lambda(\alpha{:}|\kappa|).|c| \\
|c_1 c_2| & \stackrel{\text{def}}{=} & |c_1||c_2| \\
|\,\pi_1\, c| & \stackrel{\text{def}}{=} & \pi_1\,|c| \\
|\,\pi_2\, c| & \stackrel{\text{def}}{=} & \pi_2\,|c| \\
|\langle c1, c2\rangle| & \stackrel{\text{def}}{=} & \langle |c1|, |c2|\rangle \\
|\texttt{Int}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{Int}) \\
|\,\texttt{Sum}_i(c_1,\ldots,c_n)| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{Sum}(i)[\text{interp}\,|c_1|,\ldots,\text{interp}\,|c_n|]) \\
|\texttt{Exn}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\exists(\alpha{::}\mathrm{T}_{32}).(\alpha \times (\texttt{Dyntag}\,\alpha))) \\
|\,\texttt{Dyntag}_c\,| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{Dyntag}(\text{interp}\,|c|)) \\
|\texttt{Farray}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{Array}_{64}\texttt{Float}) \\
|\texttt{Boxf}| & \stackrel{\text{def}}{=} & \mathbf{BFloat} \\
|\texttt{Unit}| & \stackrel{\text{def}}{=} & \mathbf{Record}[] \\
|c^1| & \stackrel{\text{def}}{=} & \mathbf{Record}[\text{interp}\,|c|] \\
|c_1 \times c_2]| & \stackrel{\text{def}}{=} & \mathbf{Record}[\text{interp}\,|c_1|, \text{interp}\,|c_2|] \\
|c_1 \times \ldots \times c_n| & \stackrel{\text{def}}{=} & \mathbf{Record}([\text{interp}\,|c_1|,\ldots,\text{interp}\,|c_n|]) \\
|(c_1,\ldots,c_n) \to c| & \stackrel{\text{def}}{=} & \mathbf{Other}((\text{interp}\,|c_1|,\ldots,\text{interp}\,|c_n|) \to \text{interp}\,|c|) \\
|\texttt{Array}_c| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{case}\,|c| \\
 & & \quad \mathbf{Record}\ \alpha \Rightarrow \texttt{Array}_{32}(\times(\alpha)) \\
 & & \quad \mathbf{BFloat}\ \Rightarrow \texttt{Array}_{64}(\texttt{Float}) \\
 & & \quad \mathbf{Other}\ t \Rightarrow \texttt{Array}_{32} t) \\
|\texttt{Vararg}_{c_1 \to c_2}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{case}\,|c_1| \\
 & & \quad \mathbf{Record}\,[] \Rightarrow () \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{Record}\,[t] \Rightarrow (t) \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{Record}\,[t_1,t_2] \Rightarrow (t_1,t_2) \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{Record}\ l \Rightarrow (\times(l)) \to \text{interp}\,|c_2| \\
 & & \quad \mathbf{BFloat}\ \Rightarrow (\texttt{Boxed Float}) \to \text{interp}\,|c_2|) \\
 & & \quad \mathbf{Other}\ t \Rightarrow (t) \to \text{interp}\,|c_2|) \\
|\mu(\alpha,\beta).(c_1,c_2)| & \stackrel{\text{def}}{=} &
\end{array}
$$

$$
\begin{array}{l}
\langle \mathbf{Other}(\texttt{rec}_{1+1}\langle f, \texttt{inj}_1 *\rangle), \mathbf{Other}(\texttt{rec}_{1+1}\langle f, \texttt{inj}_2 *\rangle)\rangle \\
\quad \text{where} f = \lambda(\rho{:}1+1 \to \mathrm{T}_{32}).\lambda(\omega.1+1). \\
\qquad \texttt{case}\,\omega \\
\qquad\quad \texttt{inj}_1\,\_ \Rightarrow \text{interp}(|c_1|[\rho(\texttt{inj}_1 *)/\alpha, \rho(\texttt{inj}_2 *)/\beta]) \\
\qquad\quad |\ \texttt{inj}_2\,\_ \Rightarrow \text{interp}(|c_2|[\rho(\texttt{inj}_1 *)/\alpha, \rho(\texttt{inj}_2 *)/\beta])
\end{array}
$$

Contexts $\boxed{|\Delta|}$

$$
\begin{array}{llll}
|\Delta, \alpha{:}\kappa| & \stackrel{\text{def}}{=} & \Delta', \alpha{:}|\kappa|; \Gamma & \text{if } |\Delta| = \Delta'; \Gamma \\
|\Delta, x{:}\tau| & \stackrel{\text{def}}{=} & \Delta'; \Gamma', x{:}|\tau| & \text{if } |\Delta| = \Delta'; \Gamma \\
|\Delta, x_{64}| & \stackrel{\text{def}}{=} & \Delta'; \Gamma', x{:}\texttt{Float} & \text{if } |\Delta| = \Delta'; \Gamma
\end{array}
$$

Type representations $\square$

$$
\begin{array}{lcl}
R : \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32} & \stackrel{\mathrm{def}}{=} & \lambda(\alpha{:}\mathrm{T}_{\mathrm{mil}}).\\
& & \quad \mathtt{case}\,\alpha\\
& & \qquad (\textbf{Record}\ \beta \Rightarrow\\
& & \qquad\quad (\mathtt{case}\,\beta\,([\,]\ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})\\
& & \qquad\quad + \mathtt{case}\,\beta\,([\_], \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})\\
& & \qquad\quad + \mathtt{case}\,\beta\,([\_,\_]\ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})\\
& & \qquad\quad + \mathtt{case}\,\beta\,(\_\ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})\\
& & \qquad \mid \_ \Rightarrow \mathtt{Void})\\
& & \quad + \mathtt{case}\,\alpha\,(\textbf{BFloat}\ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})\\
& & \quad + \mathtt{case}\,\alpha\,(\textbf{Other}\ \_ \Rightarrow \mathtt{Unit} \mid \_ \Rightarrow \mathtt{Void})
\end{array}
$$

$$
\begin{array}{lcl}
\downharpoonleft c{:}\mathrm{T}_{32}\downharpoonright & \stackrel{\mathrm{def}}{=} & R(|c|)\\
\downharpoonleft c{:}\kappa_1 \to \kappa_2\downharpoonright & \stackrel{\mathrm{def}}{=} & \forall[\alpha{:}|\kappa_1|](\downharpoonleft\alpha{:}\kappa_1\downharpoonright)\to \downharpoonleft c\alpha{:}\kappa_2\downharpoonright\\
\downharpoonleft c{:}\kappa_1 \times \kappa_2\downharpoonright & \stackrel{\mathrm{def}}{=} & \downharpoonleft\pi_1\, c{:}\kappa_1\downharpoonright \times \downharpoonleft\pi_2\, c{:}\kappa_2\downharpoonright
\end{array}
$$

$$
\begin{array}{lcl}
\downharpoonleft\alpha\downharpoonright & \stackrel{\mathrm{def}}{=} & x_\alpha\\
\downharpoonleft\lambda(\alpha{:}\kappa).c\downharpoonright & \stackrel{\mathrm{def}}{=} & \lambda[\alpha{:}|\kappa|](x_\alpha{:}\ \downharpoonleft\alpha{:}\kappa\downharpoonright).\ \downharpoonleft c\downharpoonright\\
\downharpoonleft c_1 c_2\downharpoonright & \stackrel{\mathrm{def}}{=} & \downharpoonleft c_1\downharpoonright\ [|c_2|](\downharpoonleft c_2\downharpoonright)\\
\downharpoonleft\pi_i\, c\downharpoonright & \stackrel{\mathrm{def}}{=} & \mathtt{select}_i\ \downharpoonleft c\downharpoonright\\
\downharpoonleft\langle c1, c2\rangle\downharpoonright & \stackrel{\mathrm{def}}{=} & \langle\downharpoonleft c_1\downharpoonright, \downharpoonleft c_2\downharpoonright\rangle\\
\downharpoonleft\mathtt{Unit}\downharpoonright & \stackrel{\mathrm{def}}{=} & \mathtt{inj}_1^{R(|\mathtt{Unit}|)}(\mathtt{inj}_1\langle\rangle)\\
\downharpoonleft c^1\downharpoonright & \stackrel{\mathrm{def}}{=} & \mathtt{inj}_1^{R(|c^1|)}(\mathtt{inj}_2(\mathtt{inj}_1\langle\rangle))\\
\downharpoonleft c^2\downharpoonright & \stackrel{\mathrm{def}}{=} & \mathtt{inj}_1^{R(|c^2|)}(\mathtt{inj}_2(\mathtt{inj}_2(\mathtt{inj}_1\langle\rangle)))\\
\downharpoonleft c^{n\geq 2}\downharpoonright & \stackrel{\mathrm{def}}{=} & \mathtt{inj}_1^{R(|c^n|)}(\mathtt{inj}_2(\mathtt{inj}_2(\mathtt{inj}_2\langle\rangle)))\\
\downharpoonleft\mathtt{Boxf}\downharpoonright & \stackrel{\mathrm{def}}{=} & \mathtt{inj}_2^{R(|\mathtt{Boxf}|)}\langle\rangle\\
\downharpoonleft c\ \mathtt{as}\ \mu(\alpha,\beta).(c_1,c_2)\downharpoonright & \stackrel{\mathrm{def}}{=} & \langle\mathtt{inj}_3^{R(|\ \pi_1\ c|)}\langle\rangle, \mathtt{inj}_3^{R(|\ \pi_2\ c|)}\langle\rangle\rangle\\
\downharpoonleft c\ \mathit{otherwise}\downharpoonright & \stackrel{\mathrm{def}}{=} & \mathtt{inj}_3^{R(|c|)}\langle\rangle
\end{array}
$$

$$
\begin{array}{lcl}
|\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n](\tau_1,\ldots,\tau_m)(k)\to\tau| & \stackrel{\mathrm{def}}{=} & \\
\quad \forall[\alpha_1{:}|\kappa_1|,\ldots,\alpha_n{:}|\kappa_n|] & & \\
\qquad (\downharpoonleft\alpha_1{:}\kappa_1\downharpoonright,\ldots \downharpoonleft\alpha_n{:}\kappa_n\downharpoonright, |\tau_1|,\ldots,|\tau_m|)(\mathtt{Float}_0,\ldots,\mathtt{Float}_{k-1})\to|\tau| & & \\
|T(c)| & \stackrel{\mathrm{def}}{=} & \mathrm{interp}\,|c|\\
|\tau_1\times\ldots\times\tau_n| & \stackrel{\mathrm{def}}{=} & \times[|\tau_1|,\ldots,|\tau_n|]
\end{array}
$$

Representation contexts $\square$

$$
\begin{array}{lcl}
\downharpoonleft\Delta,\alpha{:}\kappa\downharpoonright & \stackrel{\mathrm{def}}{=} & \downharpoonleft\Delta\downharpoonright, x_\alpha{:}\ \downharpoonleft\alpha{:}\kappa\downharpoonright\\
\downharpoonleft\Delta, x{:}\tau\downharpoonright & \stackrel{\mathrm{def}}{=} & \downharpoonleft\Delta\downharpoonright\\
\downharpoonleft\Delta, x_{64}\downharpoonright & \stackrel{\mathrm{def}}{=} & \downharpoonleft\Delta\downharpoonright
\end{array}
$$

Definitions for the term translation $\square$

$$\begin{array}{l}\textbf{typecase}\, c\,(e_1, e_2, e_3) \stackrel{\text{def}}{=} \\ \quad \texttt{case}\, \lfloor c \rfloor \\ \qquad \textbf{Record}\,\_ \Rightarrow \texttt{vcase}\,|c|\,(\textbf{Record}\,\_ \Rightarrow e_1 \mid \_ \Rightarrow \texttt{dead}\,x) \\ \qquad \mid \textbf{BFloat}\,\_ \Rightarrow \texttt{vcase}\,|c|\,(\textbf{BFloat}\,\_ \Rightarrow e_2 \mid \_ \Rightarrow \texttt{dead}\,x) \\ \qquad \mid \textbf{Other}\,\_ \Rightarrow \texttt{vcase}\,|c|\,(\textbf{Other}\,\_ \Rightarrow e_3 \mid \_ \Rightarrow \texttt{dead}\,x)\end{array}$$

Small Values

$\boxed{\Delta \vdash sv : \tau \rightsquigarrow \vec{d}\,\texttt{in}\,sv'}$

$$\frac{}{\Delta[x{:}\tau] \vdash x : \tau \rightsquigarrow \epsilon\,\texttt{in}\,x}\ \textit{var}$$

$$\frac{}{\Delta \vdash i : \texttt{Int} \rightsquigarrow \epsilon\,\texttt{in}\,i}\ \textit{int}$$

$$\frac{\Delta \vdash fv : \texttt{Float} \rightsquigarrow \vec{d}\,\texttt{in}\,fv'}{\Delta \vdash \texttt{boxf}\,fv : \texttt{Boxf} \rightsquigarrow \vec{d}\,\texttt{in box}\,fv'}\ \textit{boxf}$$

$$\frac{\Delta \vdash sv_i : \tau_i \rightsquigarrow \vec{d_i}\,\texttt{in}\,sv'_i}{\Delta \vdash \langle sv_1, \ldots, sv_n\rangle : \tau_1 \times \ldots \times \tau_n \rightsquigarrow \vec{d_1}, \ldots, \vec{d_n}\,\texttt{in}\,\langle sv'_1, \ldots, sv'_n\rangle}\ \textit{tuple}$$

$$\frac{}{\Delta \vdash \texttt{inj\_tag}^j_{\texttt{Sum}_i(\vec{c})} : \texttt{Sum}^j_i(\vec{c}) \rightsquigarrow \epsilon\,\texttt{in}\,\texttt{inj\_tag}^j_{(i,\mathrm{interp}\,|\vec{c}|)}\,\texttt{tag}_j}\ \textit{inj\_tag}$$

$$\frac{\Delta \vdash sv : c_j \rightsquigarrow \vec{d}\,\texttt{in}\,sv'}{\Delta \vdash \texttt{inj}^j_{\texttt{Sum}_i(\vec{c})}\,sv : \texttt{Sum}^j_i(\vec{c}) \rightsquigarrow \vec{d}\,\texttt{in inj}^j_{(i,\mathrm{interp}\,|\vec{c}|)}\langle \texttt{tag}_j, sv'\rangle}\ \textit{inj}$$

$$\frac{}{\Delta \vdash \texttt{exn\_tag}_c : \texttt{Dyntag}_c \rightsquigarrow \epsilon\,\texttt{in}\,\texttt{exn\_tag}_{\mathrm{interp}\,|c|}}\ \textit{exn\_tag}$$

$$\frac{\Delta \vdash sv_1 : \texttt{Dyntag}_c \rightsquigarrow \vec{d_1}\,\texttt{in}\,sv'_1 \quad \Delta \vdash sv_2 : c \rightsquigarrow \vec{d_2}\,\texttt{in}\,sv'_2}{\begin{array}{l}\Delta \vdash \texttt{inj\_dyn}_c(sv_1, sv_2) : \texttt{Exn} \rightsquigarrow \\ \quad \vec{d_1}, \vec{d_2} \\ \quad \texttt{in pack}\langle sv'_1, sv'_2\rangle\,\texttt{as}\,\exists\alpha{:}\mathrm{T}_{32}.(\texttt{Dyntag}(\alpha) \times \alpha)\,\texttt{hiding}\,\mathrm{interp}\,|c|\end{array}}\ \textit{inj\_exn}$$

$$\frac{\Delta \vdash sv : c_i[\pi_1\,\mu(\alpha,\beta)(c_1,c_2)/\alpha, \pi_2\,\mu(\alpha,\beta)(c_1,c_2)/\beta] \rightsquigarrow \vec{d}\,\texttt{in}\,sv'}{\begin{array}{r}\Delta \vdash \texttt{roll}_{\pi_i\,\mu(\alpha,\beta)(c_1,c_2)}\,sv : \pi_i\,\mu(\alpha,\beta)(c_1,c_2) \rightsquigarrow \\ \vec{d}\,\texttt{in roll}_{\mathrm{interp}\,|\,\pi_i\,\mu(\alpha,\beta)(c_1,c_2)|}\,sv'\end{array}}\ \textit{roll}$$

$$\frac{\Delta \vdash sv : \texttt{Sum}^j_{i,(\vec{c})} \rightsquigarrow \vec{d}\,\texttt{in}\,sv'}{\Delta \vdash sv : \texttt{Sum}_{i,(\vec{c})} \rightsquigarrow \vec{d}\,\texttt{in forgetknown}\,sv'}\ \textit{subsum}$$

Expressions $\boxed{\Delta \vdash e : \tau \leadsto e'}$

$$\frac{\Delta \vdash sv : \tau \leadsto \vec{d}\,\mathtt{in}\,sv'}{\Delta \vdash sv : \tau \leadsto \mathtt{let}\,\vec{d}\,\mathtt{in}\,sv'}\ sv$$

$$\frac{\Delta \vdash i : \tau' \leadsto \vec{d}\,\mathtt{in}\,i' \quad \Delta[x : \tau'] \vdash e : \tau \leadsto e'}{\Delta \vdash \mathtt{let}_\tau\,x = i\,\mathtt{in}\,e : \tau \leadsto \mathtt{let}_{|\tau|}\,\vec{d}, x = i'\,\mathtt{in}\,e'}\ i$$

$$\frac{\begin{array}{c}\Delta[f : \forall[\overrightarrow{\alpha{::}\kappa}](\vec{\tau})(|x_f|) \to \tau_r, \overrightarrow{\alpha{::}\kappa}, \overrightarrow{x{:}\tau}, \vec{x_f}] \vdash e_f : \tau_r \leadsto e'_f \\ \Delta[f : \forall[\overrightarrow{\alpha{::}\kappa}](\vec{\tau})(|x_f|) \to \tau_r] \vdash e : \tau \leadsto e'\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{let}_\tau\,\mathtt{rec}_{\tau_r}\,f[\overrightarrow{\alpha{::}\kappa}](\overrightarrow{x{:}\tau})(\vec{x_f}).e_f\,\mathtt{in}\,e : \forall[\overrightarrow{\alpha{::}\kappa}](\vec{\tau})(|x_f|) \to \tau \leadsto \\ \qquad \mathtt{let}_{|\tau|}\,\mathtt{rec}_{|\tau_r|}\,f[\overrightarrow{\alpha{::}|\kappa|}](\overrightarrow{x_\alpha : \lfloor \alpha{::}\kappa \rfloor}, \overrightarrow{x{:}\tau})(\overrightarrow{x_f{:}\mathtt{Float}}).e'_f\,\mathtt{in}\,e'\end{array}}\ rec$$

$$\frac{\begin{array}{cc}\Delta \vdash sv_1 : \mathtt{Array}_c \leadsto \vec{d_1}\,\mathtt{in}\,sv'_1 & \Delta \vdash sv_2 : \mathtt{Int} \leadsto \vec{d_2}\,\mathtt{in}\,sv'_2 \\ \Delta \vdash sv_3 : c \leadsto \vec{d_3}\,\mathtt{in}\,sv'_3 & \Delta \vdash e : \tau \leadsto e'\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{let}_\tau\,\mathtt{upd}_c(sv_1, sv_2, sv_3)\,\mathtt{in}\,e : \tau \leadsto \\ \qquad\qquad \mathtt{let}_{|\tau|}\,\vec{d_1}, \vec{d_2}, \vec{d_3}, \\ \qquad\qquad\quad \mathbf{typecase}\,c\,(\\ \qquad\qquad\qquad \mathtt{upd}_{\mathrm{interp}\,|c|}(sv'_1, sv'_2, sv'_3) \\ \qquad\qquad\qquad \mathtt{upd}_{\mathtt{Float}}(sv'_1, sv'_2, \mathtt{unbox}\,sv'_3) \\ \qquad\qquad\qquad \mathtt{upd}_{\mathrm{interp}\,|c|}(sv'_1, sv'_2, sv'_3)) \\ \qquad\qquad \mathtt{in}\,e'\end{array}}\ upd$$

$$\frac{\begin{array}{cc}\Delta \vdash sv_1 : \mathtt{Farray} \leadsto \vec{d_1}\,\mathtt{in}\,sv'_1 & \Delta \vdash sv_2 : \mathtt{Int} \leadsto \vec{d_2}\,\mathtt{in}\,sv'_2 \\ \Delta \vdash fv : \mathtt{Float} \leadsto \vec{d_f}\,\mathtt{in}\,fv' & \Delta \vdash e : \tau \leadsto e'\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{let}_\tau\,\mathtt{fupd}(sv_1, sv_2, fv)\,\mathtt{in}\,e : \tau \leadsto \\ \mathtt{let}_{|\tau|}\,\vec{d_1}, \vec{d_2}, \vec{d_f}, \mathtt{upd}_{\mathtt{Float}}(sv'_1, sv'_2, fv')\,\mathtt{in}\,e'\end{array}}\ fupd$$

$$\frac{\Delta \vdash fv : \mathtt{Float} \leadsto \vec{d}\,\mathtt{in}\,fv' \quad \Delta[x_f] \vdash e : \tau \leadsto e'}{\Delta \vdash \mathtt{let}_\tau\,x_{64} = fv\,\mathtt{in}\,e : \tau \leadsto \mathtt{let}_{|\tau|}\,\vec{d}, x_{64} = fv'\,\mathtt{in}\,e'}\ fv$$

Float values $\boxed{\Delta \vdash fv : \mathtt{Float} \leadsto \vec{d}\,\mathtt{in}\,fv'}$

$$\frac{}{\Delta \vdash r : \mathtt{Float} \leadsto \epsilon\,\mathtt{in}\,r}\ r$$

$$\frac{}{\Delta[x_f] \vdash x_f : \mathtt{Float} \leadsto \epsilon\,\mathtt{in}\,x_f}\ fvar$$

$$\frac{\Delta \vdash sv : \mathtt{Boxf} \leadsto \vec{d}\,\mathtt{in}\,sv'}{\Delta \vdash \mathtt{unboxf}\,sv : \mathtt{Float} \leadsto \vec{d}\,\mathtt{in}\ \mathtt{unbox}\,sv'}\ unbox$$

Instructions $\boxed{\Delta \vdash i : \tau \rightsquigarrow \vec{d}\,\mathtt{in}\, i'}$

$$\frac{\Delta \vdash sv : \tau \rightsquigarrow \vec{d}\,\mathtt{in}\, sv'}{\Delta \vdash sv : \tau \rightsquigarrow \vec{d}\,\mathtt{in}\, sv'}\ sv$$

$$\frac{\Delta \vdash c \equiv \pi_i\,\mu(\alpha,\beta)(c_1,c_2) : \mathrm{T}_{32} \quad \Delta \vdash sv : \pi_i\,\mu(\alpha,\beta)(c_1,c_2) \rightsquigarrow \vec{d}\,\mathtt{in}\, sv'}{\begin{array}{r}\Delta \vdash \mathtt{unroll}_c\ sv : c_i[\pi_1\,\mu(\alpha,\beta)(c_1,c_2)/\alpha, \pi_2\,\mu(\alpha,\beta)(c_1,c_2)/\beta] \rightsquigarrow \\ \vec{d}\,\mathtt{in}\ \mathtt{unroll}_{\mathrm{interp}\,|\,\pi_i\,\mu(\alpha,\beta)(c_1,c_2)|}\ sv'\end{array}}\ unroll$$

$$\frac{\Delta \vdash sv : c_1 \to c_2 \rightsquigarrow \vec{d}\,\mathtt{in}\, sv'}{\begin{array}{l}
\Delta \vdash \mathtt{vararg}_{c_1\to c_2}\ sv : \mathtt{Vararg}_{c_1\to c_2} \rightsquigarrow \\
\vec{d}, \\
f' = \mathtt{case}\ \lfloor c_1 \rfloor\ (\\
\quad \mathbf{Record}\, y \Rightarrow \mathtt{vcase}\, |c_1|\ (\\
\quad\quad \mathbf{Record}\, \alpha \Rightarrow \mathtt{case}\, y \\
\quad\quad\quad |\ \mathbf{R0}\, x \Rightarrow \mathtt{vcase}\, \alpha \\
\quad\quad\quad\quad ([] \Rightarrow \mathtt{let\ rec}\, f[]().sv'[](\langle\rangle)()\ \mathtt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \mathtt{dead}\, x) \\
\quad\quad\quad |\ \mathbf{R1}\, x \Rightarrow \mathtt{vcase}\, \alpha \\
\quad\quad\quad\quad ([\beta] \Rightarrow \mathtt{let\ rec}\, f[](y{:}\beta)().sv'[](\langle y\rangle)()\ \mathtt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \mathtt{dead}\, x) \\
\quad\quad\quad |\ \mathbf{R2}\, x \Rightarrow \mathtt{vcase}\, \alpha \\
\quad\quad\quad\quad ([\beta_1,\beta_2] \Rightarrow \mathtt{let\ rec}\, f[](y_1{:}\beta_1, y_2{:}\beta_2)().sv'[](\langle y_1, y_2\rangle)()\ \mathtt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \mathtt{dead}\, x) \\
\quad\quad\quad |\ \mathbf{BigRecord}\ _ \Rightarrow sv' \\
\quad\quad |\ _ \Rightarrow \mathtt{dead}\, y) \\
\quad |\ _ \Rightarrow sv') \\
\mathtt{in}\, f'
\end{array}}\ vararg$$

$$\frac{\Delta \vdash sv : \mathtt{Vararg}_{c_1\to c_2} \rightsquigarrow \vec{d}\,\mathtt{in}\, sv'}{\begin{array}{l}
\Delta \vdash \mathtt{onearg}_{c_1\to c_2}\ sv : c_1 \to c_2 \rightsquigarrow \\
\vec{d}, \\
f' = \mathtt{case}\ \lfloor c_1 \rfloor\ (\\
\quad \mathbf{Record}\, y \Rightarrow \mathtt{vcase}\, |c_1|\ (\\
\quad\quad \mathbf{Record}\, \alpha \Rightarrow \mathtt{case}\, y \\
\quad\quad\quad |\ \mathbf{R0}\, x \Rightarrow \mathtt{vcase}\, \alpha \\
\quad\quad\quad\quad ([] \Rightarrow \mathtt{let\ rec}\, f[](y{:}\mathtt{unit})().sv'[]()()\ \mathtt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \mathtt{dead}\, x) \\
\quad\quad\quad |\ \mathbf{R1}\, x \Rightarrow \mathtt{vcase}\, \alpha \\
\quad\quad\quad\quad ([\beta] \Rightarrow \mathtt{let\ rec}\, f[](\langle y\rangle{:} \times [\beta])().sv'[](y)()\ \mathtt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \mathtt{dead}\, x) \\
\quad\quad\quad |\ \mathbf{R2}\, x \Rightarrow \mathtt{vcase}\, \alpha \\
\quad\quad\quad\quad ([\beta_1,\beta_2] \Rightarrow \mathtt{let\ rec}\, f[](\langle y_1, y_2\rangle{:} \times [\beta_1,\beta_2])().sv'[](y_1, y_2)()\ \mathtt{in}\, f \\
\quad\quad\quad\quad |\ _ \Rightarrow \mathtt{dead}\, x) \\
\quad\quad\quad |\ \mathbf{BigRecord}\ _ \Rightarrow sv' \\
\quad\quad |\ _ \Rightarrow \mathtt{dead}\, y) \\
\quad |\ _ \Rightarrow sv') \\
\mathtt{in}\, f'
\end{array}}\ onearg$$

$$\frac{\begin{array}{c}\Delta \vdash sv : \forall[\alpha{::}\kappa_1, \ldots, \alpha{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau \rightsquigarrow \vec{d}\ \mathtt{in}\ sv' \\ \Delta \vdash sv_i : \tau_i \rightsquigarrow \vec{d_i}\ \mathtt{in}\ sv_i' \qquad \Delta \vdash fv_i : \mathtt{Float} \rightsquigarrow \vec{d_i^f}\ \mathtt{in}\ fv_i'\end{array}}{\begin{array}{l}\Delta \vdash sv[c_1, \ldots, c_n](sv_1, \ldots, sv_m)(fv_1, \ldots, fv_k) : \tau[c_1/\alpha_1, \ldots, c_n/\alpha_n] \rightsquigarrow \\ \qquad \vec{d}, \vec{d_1}, \ldots, \vec{d_m}, \vec{d_1^f}, \ldots, \vec{d_k^f}, \\ \qquad x = sv'[|c_1|, \ldots, |c_n|](\lfloor c_1 \rfloor, \ldots, \lfloor c_n \rfloor, sv_1', \ldots, sv_m')(fv_1', \ldots, fv_k') \\ \qquad \mathtt{in}\ x\end{array}}\ \mathit{app}$$

$$\frac{\Delta \vdash sv : \mathtt{Sum}_i^j(\vec{c}) \rightsquigarrow \vec{d}\ \mathtt{in}\ sv'}{\Delta \vdash \mathtt{proj}^j_{\mathtt{Sum}_i(\vec{c})}\ sv : c_j \rightsquigarrow \vec{d}, \mathtt{proj}^j_{(i, \mathrm{interp}\,|\vec{c}|)}\ sv'}\ \mathit{proj}$$

$$\frac{\Delta \vdash sv : \tau_1 \times \ldots \times \tau_n \rightsquigarrow \vec{d}\ \mathtt{in}\ sv'}{\Delta \vdash \mathtt{select}^i\ sv : \tau_i \rightsquigarrow \vec{d}\ \mathtt{in}\ \mathtt{select}^i\ sv'}\ \mathit{select}$$

$$\frac{\Delta \vdash sv : \mathtt{Sum}_i(\vec{c}) \rightsquigarrow \vec{d}\ \mathtt{in}\ sv' \quad \Delta[x_j : \mathtt{Sum}_i^j(\vec{c})] \vdash e_j : \tau \rightsquigarrow e_j'}{\Delta \vdash \mathtt{case}_\tau(sv)\,(x_1.e_1, \ldots, x_n.e_n) : \tau \rightsquigarrow \vec{d}\ \mathtt{in}\ \mathtt{case}_{\mathrm{interp}\,|\tau|}(sv')\,(x_1.e_1', \ldots, x_n.e_n')}\ \mathit{case}$$

$$\frac{\Delta \vdash e_1 : \tau \rightsquigarrow e_1' \quad \Delta[x : \mathit{Exn}] \vdash e_2 : \tau \rightsquigarrow e_2'}{\Delta \vdash \mathtt{handle}_\tau(e_1, x.e_2) : \tau \rightsquigarrow \mathtt{handle}_{\mathrm{interp}\,|\tau|}(e_1', x.e_2')}\ \mathit{handle}$$

$$\frac{\Delta \vdash sv_1 : \mathtt{Array}_c \rightsquigarrow \vec{d_1}\ \mathtt{in}\ sv_1' \quad \Delta \vdash sv_2 : \mathtt{Int} \rightsquigarrow \vec{d_2}\ \mathtt{in}\ sv_2'}{\begin{array}{l}\Delta \vdash \mathtt{sub}_c(sv_1, sv_2) : c \rightsquigarrow \vec{d_1}, \vec{d_2}, \\ \qquad x = \mathbf{typecase}\ c\,(\\ \qquad\quad \mathtt{sub}_{\mathrm{interp}\,|c|}(sv_1', sv_2') \\ \qquad\quad \mathtt{box}(\mathtt{sub}_{\mathtt{Float}}(sv_1', sv_2')) \\ \qquad\quad \mathtt{sub}_{\mathrm{interp}\,|c|}(sv_1', sv_2')) \\ \qquad \mathtt{in}\ x\end{array}}\ \mathit{sub}$$

$$\frac{\Delta \vdash sv_1 : \mathtt{Array}_c \rightsquigarrow \vec{d_1}\ \mathtt{in}\ sv_1' \quad \Delta \vdash sv_2 : \mathtt{Int} \rightsquigarrow \vec{d_2}\ \mathtt{in}\ sv_2'}{\Delta \vdash \mathtt{fsub}(sv_1, sv_2) : \tau \rightsquigarrow \vec{d_1}, \vec{d_2}\ \mathtt{in}\ \mathtt{sub}_{\mathtt{Float}}(sv_1', sv_2')}\ \mathit{fsub}$$

$$\frac{\Delta \vdash sv_1 : \mathtt{Int} \rightsquigarrow \vec{d_1}\ \mathtt{in}\ sv_1' \quad \Delta \vdash sv_2 : c \rightsquigarrow \vec{d_2}\ \mathtt{in}\ sv_2'}{\begin{array}{l}\Delta \vdash \mathtt{array}_c(sv_1, sv_2) : \mathtt{Array}_c \rightsquigarrow \\ \quad \vec{d_1}, \\ \quad \vec{d_2}, \\ \quad x = \mathbf{typecase}\ c\,(\\ \qquad \mathtt{array}_{\mathrm{interp}\,|c|}(sv_1', sv_2'), \\ \qquad \mathtt{array}_{\mathtt{Float}}(sv_1', \mathtt{unbox}\ sv_2'), \\ \qquad \mathtt{array}_{\mathrm{interp}\,|c|}(sv_1', sv_2')) \\ \quad \mathtt{in}\ x\end{array}}\ \mathit{array}$$

$$\frac{\Delta \vdash sv : \texttt{Int} \rightsquigarrow \vec{d_1}\ \texttt{in}\ sv' \quad \Delta \vdash fv : \texttt{Float} \rightsquigarrow \vec{d_2}\ \texttt{in}\ fv'}{\Delta \vdash \texttt{farray}(sv, fv) : \texttt{Farray} \rightsquigarrow \vec{d_1}, \vec{d_2}\ \texttt{in}\ \texttt{array}_{\texttt{Float}}(sv', fv')}\ farray$$

$$\frac{\Delta \vdash sv : \texttt{Exn} \rightsquigarrow \vec{d}\ \texttt{in}\ sv'}{\Delta \vdash \texttt{raise}_\tau\ sv : \tau \rightsquigarrow \vec{d}\ \texttt{in}\ \texttt{raise}_{\mathrm{interp}\,|\tau|}\ sv'}\ raise$$

$$\frac{\Delta \vdash fv : \texttt{Float} \rightsquigarrow \vec{d}\ \texttt{in}\ fv'}{\Delta \vdash fv : \texttt{Float} \rightsquigarrow \vec{d}\ \texttt{in}\ fv'}\ fv$$

$$\frac{}{\Delta \vdash \texttt{mkexntag}_c : \texttt{Dyntag}_c \rightsquigarrow \texttt{dyntag}_{\mathrm{interp}\,|c|}}\ mkexntag$$

C TALT

C.1 TALT Static semantics

Notation

$\square$

$$\begin{array}{llll}
(\tau \triangleright_{32} \sigma)[0]_{32} & \stackrel{\text{def}}{=} \tau & (\phi \triangleright_{64} \sigma)[0]_{64} & \stackrel{\text{def}}{=} \phi \\
(\tau \triangleright_{32} \sigma)[n+1]_{32} & \stackrel{\text{def}}{=} \sigma[n]_{32} & (\tau \triangleright_{32} \sigma)[n+1]_{64} & \stackrel{\text{def}}{=} \sigma[n]_{64} \\
(\phi \triangleright_{64} \sigma)[n+2]_{32} & \stackrel{\text{def}}{=} \sigma[n]_{32} & (\phi \triangleright_{64} \sigma)[n+2]_{64} & \stackrel{\text{def}}{=} \sigma[n]_{64}
\end{array}$$

$$\begin{array}{ll}
(\tau \triangleright_{32} \sigma)[0]_{32} \leftarrow \tau' & \stackrel{\text{def}}{=} \tau' \triangleright_{32} \sigma \\
(\phi \triangleright_{64} \sigma)[0]_{32} \leftarrow \tau' & \stackrel{\text{def}}{=} \tau' \triangleright_{32} ns_{32} \triangleright_{32} \sigma \\
(\phi \triangleright_{64} \sigma)[1]_{32} \leftarrow \tau' & \stackrel{\text{def}}{=} ns_{32} \triangleright_{32} \tau' \triangleright_{32} \sigma \\
(\tau \triangleright_{32} \sigma)[n+1]_{32} \leftarrow \tau' & \stackrel{\text{def}}{=} (\sigma)[n]_{32} \leftarrow \tau' \\
(\phi \triangleright_{64} \sigma)[n+2]_{32} \leftarrow \tau' & \stackrel{\text{def}}{=} (\sigma)[n]_{32} \leftarrow \tau'
\end{array}$$

$$\begin{array}{ll}
(\tau_1 \triangleright_{32} \tau_2 \triangleright_{32} \sigma)[0]_{64} \leftarrow \phi' & \stackrel{\text{def}}{=} \phi' \triangleright_{64} \sigma \\
(\tau_1 \triangleright_{32} \phi \triangleright_{64} \sigma)[0]_{64} \leftarrow \phi' & \stackrel{\text{def}}{=} \phi' \triangleright_{64} ns_{32} \triangleright_{32} \sigma \\
(\phi \triangleright_{64} \sigma)[0]_{64} \leftarrow \phi' & \stackrel{\text{def}}{=} \phi' \triangleright_{64} \sigma \\
(\phi \triangleright_{64} \sigma)[1]_{64} \leftarrow \phi' & \stackrel{\text{def}}{=} ns_{32} \triangleright_{32} (ns_{32} \triangleright_{32} \sigma)[0]_{64} \leftarrow \phi' \\
(\tau \triangleright_{32} \sigma)[n+1]_{64} \leftarrow \phi' & \stackrel{\text{def}}{=} (\sigma)[n]_{64} \leftarrow \phi' \\
(\phi \triangleright_{64} \sigma)[n+2]_{64} \leftarrow \phi' & \stackrel{\text{def}}{=} (\sigma)[n]_{64} \leftarrow \phi'
\end{array}$$

$$\begin{array}{ll}
|\epsilon| & \stackrel{\text{def}}{=} 0 \\
|\tau \triangleright_{32} \sigma| & \stackrel{\text{def}}{=} 1 + |\sigma| \\
|\phi \triangleright_{64} \sigma| & \stackrel{\text{def}}{=} 2 + |\sigma|
\end{array}$$

New well-formed kind rules $\boxed{\Delta \vdash \kappa}$

$$\frac{}{\Delta \vdash ST}$$

Well-formed Constructor $\boxed{\Delta \vdash c : \kappa}$

Constants and their kinds		
$\epsilon{:}ST$	$\triangleright_{32} : \mathrm{T}_{32} \rightarrow ST \rightarrow ST$	$\triangleright_{64} : \mathrm{T}_{64} \rightarrow ST \rightarrow ST$
$\circ{:}ST \rightarrow ST \rightarrow ST$	$\mathtt{sptr}{:}ST \rightarrow \mathrm{T}_{32}$	
$ns_{64}{:}\mathrm{T}_{64}$	$ns_{32}{:}\mathrm{T}_{32}$	$\bigvee{:}\mathrm{T}_{32}\mathtt{list} \rightarrow \mathrm{T}_{32}$

$$\frac{\Delta \vdash \Gamma}{\Delta \vdash \Gamma \rightarrow 0 : \mathrm{T}_{32}}$$

Well-formed coercion $\boxed{\Delta \vdash q : \tau \Rightarrow \tau'}$

$$\frac{\Delta \vdash \tau \equiv \mathtt{rec}[\kappa](c)(c_p) : \mathrm{T}_{32}}{\Delta \vdash \mathtt{roll}_\tau : (c(\mathtt{Rec}[\kappa]c)c_p) \Rightarrow \tau}$$

$$\frac{\Delta \vdash \tau \equiv \mathtt{rec}[\kappa](c)(c_p) : \mathrm{T}_{32}}{\Delta \vdash \mathtt{unroll}_\tau\, w : (c(\mathtt{Rec}[\kappa]c)c_p) \Rightarrow \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \tau_0 :: \cdots :: \mathtt{Tag}(\overline{j}) :: \cdots :: \tau_{k-1} :: \cdots :: \tau_n :: \mathtt{nil} : \mathrm{T}_{32}\,\mathtt{list} \\ \Delta \vdash \tau_m \equiv \mathtt{Tag}(\overline{i_m}) : \mathrm{T}_{32} \quad (m = 1 \ldots k-1) \\ \Delta \vdash \tau_m \equiv \times[\mathtt{Tag}(\overline{i_m}), \tau'_m] : \mathrm{T}_{32} \quad (m = k-1 \ldots n) \\ \text{where } i_0 \ldots i_n \neq i \text{ and } 0 \leq j < i\end{array}}{\Delta \vdash \mathtt{inj\_union}^j_{(i,c)} : \mathtt{Tag}(\overline{j}) \Rightarrow \bigvee c}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \tau_0 :: \cdots :: \tau_{k-1} :: \cdots :: \times[\mathtt{Tag}(\overline{j}), \tau] :: \cdots :: \tau_n :: \mathtt{nil} : \mathrm{T}_{32}\,\mathtt{list} \\ \Delta \vdash \tau_m \equiv \mathtt{Tag}(\overline{i_m}) : \mathrm{T}_{32} \quad (m = 1 \ldots k-1) \\ \Delta \vdash \tau_m \equiv \times[\mathtt{Tag}(\overline{i_m}), \tau'_m] : \mathrm{T}_{32} \quad (m = k-1 \ldots n) \\ \text{where } i_0 \ldots i_n \neq i \text{ and } i \leq j\end{array}}{\Delta \vdash \mathtt{inj\_union}^j_{(i,c)} : \times[\mathtt{Tag}(\overline{j}), \tau] \Rightarrow \bigvee(c)}$$

$$\frac{\Delta \vdash \tau \equiv \exists[\kappa](c') : \mathrm{T}_{32} \quad \Delta \vdash c : \kappa}{\Delta \vdash \mathtt{pack}[\tau]c : (c'\, c) \Rightarrow \tau}$$

$$\frac{\Delta \vdash \tau : \mathrm{T}_{32}}{\Delta \vdash \mathtt{inj\_dyn}_\tau : (\times[\mathtt{Dyntag}(\tau), \tau]) \Rightarrow \mathtt{Dyn}}$$

Well-formed stack $\boxed{\Psi; \Delta \vdash s : \sigma}$

$$\frac{}{\Psi; \Delta \vdash \epsilon : \epsilon} \qquad \frac{\Psi; \Delta \vdash w : \tau \quad \Psi; \Delta \vdash s : \sigma}{\Psi; \Delta \vdash w \triangleright_{32} s : \tau \triangleright_{32} \sigma} \qquad \frac{\Psi; \Delta \vdash l : \phi \quad \Psi; \Delta \vdash s : \sigma}{\Psi; \Delta \vdash l \triangleright_{32} s : \phi \triangleright_{64} \sigma}$$

Well-formed 64-bit value $\boxed{\Psi; \Delta \vdash l : \phi}$

$$\frac{}{\Psi; \Delta \vdash ns_{64} : ns_{64}}$$

$$\frac{}{\Psi; \Delta \vdash \mathtt{r} : \mathtt{Float}} \qquad \frac{}{\Psi, \ell{:}\phi, \Psi'; \Delta \vdash \ell : \phi}$$

Well-formed 32-bit value $\boxed{\Psi; \Delta \vdash w : \tau}$

$$\frac{}{\Psi, \ell{:}\tau\Psi'; \Delta \vdash \ell : \tau} \qquad \frac{}{\Psi; \Delta \vdash i : \mathtt{Int}} \qquad \frac{}{\Psi; \Delta \vdash ns_{32} : ns_{32}}$$

$$\frac{}{\Psi; \Delta \vdash \mathtt{tag}_i : \mathtt{Tag}(\bar{i})} \qquad \frac{\Psi; \Delta \vdash w : \forall[\kappa](c') \quad \Delta \vdash c : \kappa}{\Psi; \Delta \vdash w[c] : c'c}$$

$$\frac{\Psi; \Delta \vdash \sigma \equiv \sigma' : ST \quad (|\sigma'| = i)}{\Psi; \Delta \vdash \mathtt{sptr}(i) : \mathtt{sptr}(\sigma)} \qquad \frac{\Delta \vdash q : \tau_1 \Rightarrow \tau_2 \quad \Psi; \Delta \vdash w : \tau_1}{\Psi; \Delta \vdash q\, w : \tau_2}$$

Well-formed 64-bit operand $\boxed{\Psi; \Delta; \Gamma \vdash fv : \phi}$

$$\frac{\Gamma(f) = \phi}{\Psi; \Delta; \Gamma \vdash f : \phi} \qquad \frac{\Psi; \Delta \vdash l : \phi}{\Psi; \Delta; \Gamma \vdash l : \phi} \qquad \frac{\Gamma(\mathbf{sp}) = \sigma \quad \sigma[i]_{64} = \phi}{\Psi; \Delta; \Gamma \vdash \mathbf{sp}(i) : \phi}$$

Well-formed 32-bit operand $\boxed{\Psi; \Delta; \Gamma \vdash sv : \tau}$

$$\frac{\Gamma(r) = \tau}{\Psi; \Delta; \Gamma \vdash r : \tau} \qquad \frac{\Psi; \Delta \vdash w : \tau}{\Psi; \Delta; \Gamma \vdash w : \tau} \qquad \frac{\Gamma(\mathbf{sp}) = \sigma \quad \sigma[i]_{32} = \tau}{\Psi; \Delta; \Gamma \vdash \mathbf{sp}(i) : \tau}$$

$$\frac{\Psi; \Delta; \Gamma \vdash sv : \forall[\kappa](c') \quad \Delta \vdash c : \kappa}{\Psi; \Delta; \Gamma \vdash sv[c] : c'c} \qquad \frac{\Delta \vdash q : \tau_1 \Rightarrow \tau_2 \quad \Psi; \Delta; \Gamma \vdash sv : \tau_1}{\Psi; \Delta; \Gamma \vdash q\, sv : \tau_2}$$

Well-formed instruction $\boxed{\Psi; \Delta; \Gamma \vdash i \Rightarrow \Delta; \Gamma}$

$$\frac{\Psi; \Delta; \Gamma \vdash sv : \tau}{\Psi; \Delta; \Gamma \vdash \texttt{mov}\, \mathbf{r}, sv \Rightarrow \Delta; \Gamma\{\mathbf{r}{:}\tau\}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash \mathbf{r}_s : \times[\tau_0, \ldots, \tau_i, \ldots, \tau_n]}{\Psi; \Delta; \Gamma \vdash \texttt{loadr}\, \mathbf{r}_d, \mathbf{r}_s(i) \Rightarrow \Delta; \Gamma\{\mathbf{r}_d{:}\tau_i\}}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash sv : \tau_i \\ \Psi; \Delta; \Gamma \vdash \mathbf{r}_s : \times[\tau_0, \ldots, \tau_i, \ldots, \tau_n]\end{array}}{\Psi; \Delta; \Gamma \vdash \texttt{store}\, \mathbf{r}_s(i), sv \Rightarrow \Delta; \Gamma}$$

$$\frac{\Psi; \Delta; \Gamma \vdash sv_i : \tau_i}{\Psi; \Delta; \Gamma \vdash \texttt{malloc}\, \mathbf{r}[\tau_1, \ldots, \tau_n]\langle sv_1, \ldots, sv_n\rangle \Rightarrow \Delta; \Gamma\{\mathbf{r}{:} \times[\tau_1, \ldots, \tau_n]\}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash sv_1 : \texttt{Int} \quad \Psi; \Delta; \Gamma \vdash sv_2 : \tau}{\Psi; \Delta; \Gamma \vdash \texttt{malloc}_\tau\, \mathbf{r}(sv_1, sv_2) \Rightarrow \Delta; \Gamma\{\mathbf{r}{:}\texttt{Array}_{32}(\tau)\}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash sv : \texttt{Int} \quad \Psi; \Delta; \Gamma \vdash fv : \phi}{\Psi; \Delta; \Gamma \vdash \texttt{malloc}_\phi\, \mathbf{r}(sv, fv) \Rightarrow \Delta; \Gamma\{\mathbf{r}{:}\texttt{Array}_{64}(\phi)\}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash fv : \phi}{\Psi; \Delta; \Gamma \vdash \texttt{malloc}_\phi\, \mathbf{r}, fv \Rightarrow \Delta; \Gamma\{\mathbf{r}{:}\, \texttt{Boxed}(\phi)\}}$$

$$\frac{\Psi; \Delta; \Gamma \vdash sv : \Gamma\{\mathbf{sp}{:}(\Gamma_{\mathbf{ret}} \to 0) \triangleright_{32} \sigma\} \to 0 \quad (\Gamma(\mathbf{sp}) = \sigma)}{\Psi; \Delta; \Gamma \vdash \texttt{call}\, sv \Rightarrow \Delta; \Gamma_{\mathbf{ret}}}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash sv : \Gamma\{\mathbf{r}{:} \bigvee[\texttt{Tag}(\bar{i})]\} \to 0 \\ \Delta \vdash \Gamma(\mathbf{r}) \equiv \bigvee(\tau_0 :: \cdots :: \tau_{k-1} :: \texttt{Tag}(\bar{i}) :: c) : \mathrm{T}_{32}\end{array}}{\Psi; \Delta; \Gamma \vdash \texttt{brtag}_i\, \mathbf{r}, sv \Rightarrow \Delta; \Gamma\{\mathbf{r}{:} \bigvee(\tau_0 :: \cdots :: \tau_{k-1} :: c)\}}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash sv : \Gamma\{\mathbf{r}{:} \bigvee[\times[\texttt{Tag}(\bar{i}), \tau]]\} \to 0 \\ \Delta \vdash \Gamma(\mathbf{r}) \equiv \bigvee(\tau_0 :: \cdots :: \tau_{k-1} :: \tau_k :: \cdots \tau_n :: (\times[\texttt{Tag}(\bar{i}), \tau]) :: c) : \mathrm{T}_{32}\end{array}}{\Psi; \Delta; \Gamma \vdash \texttt{brtgd}_i\, \mathbf{r}, sv \Rightarrow \Delta; \Gamma\{\mathbf{r}{:} \bigvee(\tau_0 :: \cdots :: \tau_n :: c)\}}$$

$$\frac{}{\Psi; \Delta; \Gamma \vdash \texttt{dyntag}_\tau\, \mathbf{r} \Rightarrow \Delta; \Gamma\{\mathbf{r}{:}\, \texttt{Dyntag}(\tau)\}}$$

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma \vdash sv : \tau \quad \Gamma(\mathbf{sp}) = \sigma \\ \Delta \vdash \sigma \equiv \sigma' : ST \quad (\sigma')[i]_{32} \leftarrow \tau = \sigma''\end{array}}{\Psi; \Delta; \Gamma \vdash \texttt{swrite}\, \mathbf{sp}(i), sv \Rightarrow \Delta; \Gamma\{\mathbf{sp}{:}\sigma''\}}$$

$$\frac{}{\Psi;\Delta;\Gamma \vdash \mathtt{salloc}\, n \Rightarrow \Delta;\Gamma\{\mathbf{sp}{:}\, \underbrace{ns_{32} \rhd_{32} \cdots \rhd_{32} ns_{32}}_{n} \rhd_{32} \Gamma(\mathbf{sp})\}}$$

$$\frac{\Gamma(\mathbf{sp}) = \sigma \quad \Delta \vdash \sigma \equiv \sigma_1 \circ \sigma_2 : ST \quad |\sigma_1| = n}{\Psi;\Delta;\Gamma \vdash \mathtt{sfree}\, n \Rightarrow \Delta;\Gamma\{\mathbf{sp}{:}\sigma_2\}}$$

$$\frac{\Gamma(\mathbf{sp}) = \sigma \quad \Delta \vdash \sigma \equiv \sigma_1 \circ (\phi \rhd_{64} \sigma_2) : ST \quad |\sigma_1| = (n-1)}{\Psi;\Delta;\Gamma \vdash \mathtt{sfree}\, n \Rightarrow \Delta;\Gamma\{\mathbf{sp}{:}ns_{32} \rhd_{32} \sigma_2\}}$$

$$\frac{\Gamma(\mathbf{sp}) = \sigma}{\Psi;\Delta;\Gamma \vdash \mathtt{mov}\, \mathbf{r}, \mathbf{sp} \Rightarrow \Delta;\Gamma\{\mathbf{r}{:}\, \mathtt{sptr}(\sigma)\}}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash sv : \mathtt{sptr}(\sigma_1) \\ \Delta \vdash \Gamma(\mathbf{sp}) \equiv \sigma_2 \circ \sigma_2 : ST\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{mov}\, \mathbf{sp}, sv \Rightarrow \Delta;\Gamma\{\mathbf{sp}{:}\sigma_2\}}$$

$$\frac{\Psi;\Delta;\Gamma \vdash sv : \exists[\kappa](c)}{\Psi;\Delta;\Gamma \vdash \mathtt{unpack}[\alpha, \mathbf{r}], sv \Rightarrow \Delta, \alpha{:}\kappa;\Gamma\{\mathbf{r}{:}(c\,\alpha)\}}\ \alpha \notin \Delta$$

$$\frac{\Psi;\Delta;\Gamma \vdash sv_1 : \mathtt{Array}_{32}(\tau) \quad \Psi;\Delta;\Gamma \vdash sv_2 : \mathtt{Int}}{\Psi;\Delta;\Gamma \vdash \mathtt{sub}_\tau\, \mathbf{r}, sv_1, sv_2 \Rightarrow \Delta;\Gamma\{\mathbf{r}{:}\tau\}}$$

$$\frac{\Psi;\Delta;\Gamma \vdash sv_1 : \mathtt{Array}_{32}(\tau) \quad \Psi;\Delta;\Gamma \vdash sv_2 : \mathtt{Int} \quad \Psi;\Delta;\Gamma \vdash sv_3 : \tau}{\Psi;\Delta;\Gamma \vdash \mathtt{upd}_\tau\, sv_1, sv_2, sv_3 \Rightarrow \Delta;\Gamma}$$

$$\frac{\Psi;\Delta;\Gamma \vdash fv : \phi}{\Psi;\Delta;\Gamma \vdash \mathtt{fmov}\, \mathbf{f}, fv \Rightarrow \Delta;\Gamma\{\mathbf{f}{:}\phi\}}$$

$$\frac{\Psi;\Delta;\Gamma \vdash \mathbf{r}_s : \mathtt{Boxed}(\phi)}{\Psi;\Delta;\Gamma \vdash \mathtt{floadr}\, \mathbf{f}_d, \mathbf{r}_s \Rightarrow \Delta;\Gamma\{\mathbf{f}_d{:}\phi\}} \qquad \frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash fv : \phi \\ \Psi;\Delta;\Gamma \vdash \mathbf{r} : \mathtt{Boxed}(\phi)\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{store}\, \mathbf{r}, fv \Rightarrow \Delta;\Gamma}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash fv : \phi \quad \Gamma(\mathbf{sp}) = \sigma \\ \Delta \vdash \sigma \equiv \sigma' : ST \quad (\sigma')[i]_{64} \leftarrow \phi = \sigma''\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{fswrite}\, \mathbf{sp}(i), fv \Rightarrow \Delta;\Gamma\{\mathbf{sp}{:}\sigma''\}}$$

$$\frac{\Psi;\Delta;\Gamma \vdash sv_1 : \mathtt{Array}_{64}(\tau) \quad \Psi;\Delta;\Gamma \vdash sv_2 : \mathtt{Int}}{\Psi;\Delta;\Gamma \vdash \mathtt{sub}_\phi\, \mathbf{f}, sv_1, sv_2 \Rightarrow \Delta;\Gamma\{\mathbf{f}{:}\phi\}}$$

$$\frac{\Psi;\Delta;\Gamma \vdash sv_1 : \mathtt{Array}_{64}(\tau) \quad \Psi;\Delta;\Gamma \vdash sv_2 : \mathtt{Int} \quad \Psi;\Delta;\Gamma \vdash fv : \phi}{\Psi;\Delta;\Gamma \vdash \mathtt{upd}_\phi\, sv_1, sv_2, fv \Rightarrow \Delta;\Gamma}$$

Well-formed Instruction Sequence $\boxed{\Psi;\Delta;\Gamma \vdash I : \tau}$

$$\frac{\begin{array}{c}\Gamma(\mathbf{sp}) = (\Gamma_{\mathtt{ret}} \to 0) \rhd_{32} \sigma \\ \Gamma_{\mathtt{ret}} = \Gamma\{\mathbf{sp}{:}\sigma\}\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{ret}} \qquad \frac{\Psi;\Delta;\Gamma \vdash sv : \Gamma \to 0}{\Psi;\Delta;\Gamma \vdash \mathtt{jmp}\, sv}$$

$$\frac{\Psi;\Delta;\Gamma \vdash \mathbf{r}_t : \tau}{\Psi;\Delta;\Gamma \vdash \mathtt{halt}_\tau} \qquad \frac{\Psi;\Delta;\Gamma \vdash i \Rightarrow \Delta';\Gamma' \quad \Psi;\Delta';\Gamma' \vdash I}{\Psi;\Delta;\Gamma \vdash i; I}$$

$$\frac{\begin{array}{c}\Psi[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha];\Delta,\alpha_1{:}\kappa_1,\Delta';\Gamma[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha] \vdash sv[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha] : \mathtt{Void} \\ \Psi[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha];\Delta,\alpha_2{:}\kappa_2,\Delta';\Gamma[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha] \vdash I[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha] \\ \Delta,\alpha{:}\kappa_1+\kappa_2,\Delta';\Gamma \vdash c \equiv \alpha : \kappa_1+\kappa_2 \quad \alpha_1,\alpha_2 \notin \Delta,\Delta'\end{array}}{\Psi;\Delta,\alpha{:}\kappa_1+\kappa_2,\Delta';\Gamma \vdash \mathtt{vcase}[\alpha_1\,.\,\mathtt{dead}\,sv,\alpha_2]c; I}$$

$$\frac{\begin{array}{c}\Psi[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha];\Delta,\alpha_1{:}\kappa_1,\Delta';\Gamma[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha] \vdash I[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha] \\ \Psi[\mathtt{inj}_1^{\kappa_1+\kappa_2}\,\alpha_1/\alpha];\Delta,\alpha_2{:}\kappa_2,\Delta';\Gamma[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha] \vdash sv[\mathtt{inj}_2^{\kappa_1+\kappa_2}\,\alpha_2/\alpha] : \mathtt{Void} \\ \Delta,\alpha{:}\kappa_1+\kappa_2,\Delta' \vdash c \equiv \alpha : \kappa_1+\kappa_2 \quad \alpha_1,\alpha_2 \notin \Delta,\Delta'\end{array}}{\Psi;\Delta,\alpha{:}\kappa_1+\kappa_2,\Delta' \vdash \mathtt{vcase}[\alpha_1,\alpha_2\,.\,\mathtt{dead}\,sv]c; I}$$

$$\frac{\Delta \vdash c \equiv \mathtt{inj}_2^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \quad \Psi;\Delta;\Gamma \vdash I[c'/\alpha_2]}{\Psi;\Delta;\Gamma \vdash \mathtt{vcase}[\alpha_1\,.\,\mathtt{dead}\,sv,\alpha_2]c; I)}$$

$$\frac{\Delta \vdash c \equiv \mathtt{inj}_1^{\kappa_1+\kappa_2}\,c' : \kappa_1+\kappa_2 \quad \Psi;\Delta;\Gamma \vdash I[c'/\alpha_1]}{\Psi;\Delta;\Gamma \vdash \mathtt{vcase}[\alpha_1,\alpha_2 . sv]c; I}$$

$$\frac{\begin{array}{c}\Psi[\langle\beta,\gamma\rangle/\alpha];\Delta,\beta{:}\kappa_1,\gamma{:}\kappa_2,\Delta';\Gamma[\langle\beta,\gamma\rangle/\alpha] \vdash I[\langle\beta,\gamma\rangle/\alpha] \\ \Delta,\alpha{:}\kappa_1\times\kappa_2,\Delta' \vdash c \equiv \alpha : \kappa_1\times\kappa_2\end{array}}{\Psi;\Delta,\alpha{:}\kappa_1\times\kappa_2,\Delta';\Gamma \vdash \mathtt{refine}[\langle\beta,\gamma\rangle]\,c; I}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash I[c_1,c_2/\beta,\gamma] \\ \Delta \vdash c \equiv \langle c_1,c_2\rangle : \kappa_1\times\kappa_2\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{refine}[\langle\beta,\gamma\rangle]\,c; I}$$

$$\frac{\begin{array}{c}\Psi[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha];\Delta,\beta{:}\kappa[\mu j.\kappa/j],\Delta';\Gamma[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] \vdash I[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] \\ \Delta,\alpha{:}\mu j.\kappa,\Delta' \vdash c \equiv \alpha : \mu j.\kappa\end{array}}{\Psi;\Delta,\alpha{:}\mu j.\kappa,\Delta';\Gamma \vdash \mathtt{refine}[\mathtt{fold}\,\beta]\,c; I}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash I[c'/\beta] \\ \Delta \vdash c \equiv \mathtt{fold}_{\mu j.\kappa}\,c' : \mu j.\kappa\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{refine}[\mathtt{fold}\,\beta]\,c; I}$$

Well-formed *hval* $\boxed{\Psi\Delta;\Gamma \vdash hval : \tau \ \ \textbf{hval}}$

$$\frac{\Psi;\bullet \vdash w_i : \tau_i \quad i = 1 \ldots n}{\Psi \vdash \langle w_1,\ldots,w_n\rangle : \times[\tau_1,\ldots,\tau_n] \ \ \textbf{hval}}$$

$$\frac{\Psi;\bullet \vdash w_i : \tau \quad i = 1 \ldots n}{\Psi \vdash [w_1,\ldots,w_n] : \texttt{Array}_{32}\tau \ \ \textbf{hval}}$$

$$\frac{\Psi;\bullet \vdash l : \phi}{\Psi \vdash l : \texttt{Boxed}\,\phi \ \ \textbf{hval}} \qquad \frac{\Psi;\bullet \vdash l_i : \phi \quad i = 1 \ldots n}{\Psi \vdash [l_1,\ldots,l_n] : \texttt{Array}_{64}\phi \ \ \textbf{hval}}$$

$$\frac{\bullet \vdash \tau : \mathrm{T}_{32}}{\Psi;\bullet \vdash \texttt{dtag} : \texttt{Dyntag}_\tau \ \ \textbf{hval}}$$

$$\frac{\begin{array}{c}\alpha_0{:}\kappa_0,\ldots,\alpha_{i-1}{:}\kappa_{i-1} \vdash \kappa_i \quad \vec{\alpha}{:}\vec{\kappa} \vdash \Gamma \\ \Psi;\overrightarrow{\alpha}{:}\vec{\kappa};\Gamma \vdash I\end{array}}{\Psi \vdash \texttt{code}_\tau[\overrightarrow{\alpha{:}\kappa}]\Gamma.I : \forall[\vec{\alpha}{:}\vec{\kappa}]\Gamma \to 0 \ \ \textbf{hval}}$$

Well-formed Heap $\boxed{\vdash H : \Psi}$

$$\frac{\vdash \Psi \quad \Psi \vdash hval_i : \tau_i \ \ \textbf{hval}}{\vdash \{\ell_1{:}\tau_1 \mapsto hval_1,\ldots,\ell_n{:}\tau_n \mapsto hval_n\} : \Psi} \ \Psi = \{\ell_1{:}\tau_1,\ldots,\ell_n{:}\tau_n\}$$

Well-formed Register File $\boxed{\Psi \vdash R : \Gamma}$

$$\frac{\begin{array}{c}\Psi;\bullet \vdash w_1 : \tau_1 \quad \Psi;\bullet \vdash w_2 : \tau_2 \quad \Psi;\bullet \vdash w_e : \tau_e \quad \Psi;\bullet \vdash w_t : \tau_t \\ \Psi;\bullet \vdash l_1 : \phi_1 \quad \Psi;\bullet \vdash l_2 : \phi_2 \quad \Psi;\bullet \vdash s : \sigma\end{array}}{\begin{array}{r}\Psi \vdash \{\mathbf{r}_1 \mapsto w_1, \mathbf{r}_2 \mapsto w_2, \mathbf{r}_e \mapsto w_e, \mathbf{r}_t \mapsto w_t, \mathbf{f}_1 \mapsto l_1, \mathbf{f}_2 \mapsto l_2{:}\mathbf{sp} \mapsto s\} \\ : \{\mathbf{r}_1{:}\tau_1, \mathbf{r}_2{:}\tau_2, \mathbf{r}_e{:}\tau_e, \mathbf{r}_t{:}\tau_t, \mathbf{f}_1{:}\phi_1, \mathbf{f}_2{:}\phi_2, \mathbf{sp} : \sigma\}\end{array}}$$

Well-formed Program $\boxed{\vdash (H,R,I)}$

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi;\bullet;\Gamma \vdash I}{\vdash (H,R,I)}$$

C.2 TALT dynamic semantics

C.2.1 Definitions

$$
\begin{array}{llll}
(w \triangleright_{32} s)[0]_{32} & \stackrel{\text{def}}{=} w & (l \triangleright_{64} s)[0]_{64} & \stackrel{\text{def}}{=} l \\
(w \triangleright_{32} s)[n+1]_{32} & \stackrel{\text{def}}{=} s[n]_{32} & (w \triangleright_{32} s)[n+1]_{64} & \stackrel{\text{def}}{=} s[n]_{64} \\
(l \triangleright_{64} s)[n+2]_{32} & \stackrel{\text{def}}{=} s[n]_{32} & (l \triangleright_{64} s)[n+2]_{64} & \stackrel{\text{def}}{=} s[n]_{64}
\end{array}
$$

$$
\begin{array}{ll}
(w \triangleright_{32} s)[0]_{32} \leftarrow w' & \stackrel{\text{def}}{=} w' \triangleright_{32} s \\
(l \triangleright_{64} s)[0]_{32} \leftarrow w' & \stackrel{\text{def}}{=} w' \triangleright_{32} ns_{32} \triangleright_{32} s \\
(l \triangleright_{64} s)[1]_{32} \leftarrow w' & \stackrel{\text{def}}{=} ns_{32} \triangleright_{32} w' \triangleright_{32} s \\
(w \triangleright_{32} s)[n+1]_{32} \leftarrow w' & \stackrel{\text{def}}{=} (s)[n]_{32} \leftarrow w' \\
(l \triangleright_{64} s)[n+2]_{32} \leftarrow w' & \stackrel{\text{def}}{=} (s)[n]_{32} \leftarrow w'
\end{array}
$$

$$
\begin{array}{ll}
(w_1 \triangleright_{32} w_2 \triangleright_{32} s)[0]_{64} \leftarrow l' & \stackrel{\text{def}}{=} l' \triangleright_{64} s \\
(w_1 \triangleright_{32} l \triangleright_{64} s)[0]_{64} \leftarrow l' & \stackrel{\text{def}}{=} l' \triangleright_{64} ns_{32} \triangleright_{32} s \\
(l \triangleright_{64} s)[0]_{64} \leftarrow l' & \stackrel{\text{def}}{=} l' \triangleright_{64} s \\
(l \triangleright_{64} s)[1]_{64} \leftarrow l' & \stackrel{\text{def}}{=} ns_{32} \triangleright_{32} (ns_{32} \triangleright_{32} s)[0]_{64} \leftarrow l' \\
(w \triangleright_{32} s)[n+1]_{64} \leftarrow l' & \stackrel{\text{def}}{=} (s)[n]_{64} \leftarrow l' \\
(l \triangleright_{64} s)[n+2]_{64} \leftarrow l' & \stackrel{\text{def}}{=} (s)[n]_{64} \leftarrow l'
\end{array}
$$

$$
\hat{R}(sv) \stackrel{\text{def}}{=} \begin{cases}
R(r) & \text{when } sv = r \\
R(\mathbf{sp})[i]_{32} & \text{when } sv = \mathbf{sp}(i) \\
w & \text{when } sv = w \\
\mathtt{tag}_i & \text{when } sv = \mathtt{tag}_i \\
\hat{R}(sv')[\vec{c}] & \text{when } sv = sv'[\vec{c}] \\
q\ \hat{R}(sv') & \text{when } sv = q\ sv' \\
\mathtt{pack}\, \hat{R}(sv')\ \mathtt{as}\ \tau\ \mathtt{hiding}\, c & \text{when } sv = \mathtt{pack}\, sv'\ \mathtt{as}\ \tau\ \mathtt{hiding}\, c
\end{cases}
$$

$$
\hat{R}(fv) \stackrel{\text{def}}{=} \begin{cases}
R(f) & \text{when } fv = f \\
R(\mathbf{sp})[i]_{64} & \text{when } fv = \mathbf{sp}(i)
\end{cases}
$$

C.3 Transitions

$(H, R, I) \longmapsto P$ where	
if $I =$	then $P =$
$\texttt{mov}\,\mathbf{r}, sv; I'$	$(H, R\{\mathbf{r} \mapsto \hat{R}(sv)\}, I')$
$\texttt{loadr}\,\mathbf{r}_d, \mathbf{r}_s(i); I'$	$(H, R\{\mathbf{r} \mapsto w_i\}, I')$ where $R(\mathbf{r}_d) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_{n-1}\rangle$
$\texttt{store}\,\mathbf{r}_d(i), sv; I'$	$(H\{\ell \mapsto \langle w_0, \ldots, \hat{R}(sv), \ldots, w_{n-1}\rangle\}, R, I')$ where $R(\mathbf{r}_d) = \ell$ and $H(\ell) = \langle w_0, \ldots, w_i, \ldots, w_{n-1}\rangle$
$\texttt{malloc}\,\mathbf{r}[\tau_1, \ldots, \tau_n]\langle sv_0, \ldots, sv_{n-1}\rangle; I'$	$(H\{\ell \mapsto \langle \hat{R}(sv_0), \ldots, \hat{R}(sv_{n-1})\rangle\}, R\{\mathbf{r}_d \mapsto \ell\}, I')$ where $\ell \notin H$
$\texttt{malloc}_\tau\,\mathbf{r}_d(sv_1, sv_2); I'$	$(H\{\ell \mapsto [\underbrace{\hat{R}(sv_2), \ldots, \hat{R}(sv_2)}_{\hat{R}(sv_1)}]\}, R\{\mathbf{r}_d \mapsto \ell\}, I')$ where $\ell \notin H$
$\texttt{malloc}_\phi\,\mathbf{r}, fv; I'$	$(H\{\ell \mapsto \hat{R}(fv)\}, R\{\mathbf{r}_d \mapsto \ell\}, I')$ where $\ell \notin H$
$\texttt{malloc}_\phi\,\mathbf{r}(sv, fv); I'$	$(H\{\ell \mapsto [\underbrace{\hat{R}(fv), \ldots, \hat{R}(fv)}_{\hat{R}(sv)}]\}, R\{\mathbf{r}_d \mapsto \ell\}, I')$ where $\ell \notin H$
$\texttt{dyntag}_c\,\mathbf{r}; I'$	$(H\{\ell \mapsto \texttt{exn\_tag}\}, R\{\mathbf{r} \mapsto \ell\}, I')$ where $\ell \notin H$
$\texttt{swrite}\,\mathbf{sp}(i), sv; I'$	$(H, R\{\mathbf{sp} \mapsto s'\}, I')$ where $s' = (R(\mathbf{sp}))[i]_{32} \leftarrow \hat{R}(sv)$
$\texttt{salloc}\,n; I'$	$(H, R\{\mathbf{sp} \mapsto \underbrace{ns_{32} \triangleright_{32} \cdots \triangleright_{32} ns_{32}}_{n} \triangleright_{32} R(\mathbf{sp})\}, I')$
$\texttt{sfree}\,n; I'$	$(H, R\{\mathbf{sp} \mapsto s'\}, I')$ where $\texttt{pop}(R(\mathbf{sp}), n) = s'$
$\texttt{mov}\,\mathbf{r}, \mathbf{sp}; I'$	$(H, R\{\mathbf{r} \mapsto \texttt{sptr}(\lvert R(\mathbf{sp})\rvert)\}, I')$
$\texttt{mov}\,\mathbf{sp}, sv; I'$	$(H, R\{\mathbf{sp} \mapsto s'\}, I')$ where $R(\mathbf{sp}) = s$ and $\hat{R}(sv) = \texttt{sptr}(j) = \lvert s'\rvert$ and s' is a suffix of s

$(H, R, I) \longmapsto P$ where	
$\texttt{call}_\Gamma\ sv; I'$	$(H\{\ell \mapsto \texttt{code}[].\Gamma.I'\}, R\{\mathbf{sp} \mapsto \ell \triangleright_{32} \hat{R}(\mathbf{sp})\}, I_d[\vec{c}/\Delta])$ where $\hat{R}(sv) = \ell_d[\vec{c}]$ and $H(\ell_d) = \texttt{code}[\Delta]\Gamma.I_d$ and $\ell \notin H$
$\texttt{brtag}_i\ \mathbf{r}, sv; I'$ when $R(r) = \texttt{inj\_union}^i_{(c_1,c_2)}\ \texttt{tag}_i$	$(H, R\{\mathbf{r} \mapsto \texttt{inj\_union}^i_{([\texttt{Tag}(\bar{i})],\texttt{nil})}\ \texttt{tag}_i\}, I_d)$ where $\hat{R}(sv) = \ell$ and $H(\ell) = \texttt{code}[]\Gamma.I_d$
$\texttt{brtag}_i\ \mathbf{r}, sv; I'$ when $R(r) \neq \texttt{inj\_union}^i_{(c_1,c_2)}\ \texttt{tag}_i$	$(H, R\{\mathbf{r} \mapsto \texttt{inj\_union}^i_{(c_1',c_2)}\ \texttt{tag}_i\}, I')$ where c_1 normalizes to $\tau_0 :: \cdots :: \tau_k :: \cdots \tau_n$ and $\tau_k = \texttt{Tag}(\bar{i})$ and $c_1' = \tau_0 :: \cdots :: \tau_{k-1} :: \tau_{k+1} :: \cdots \tau_n$
$\texttt{brtgd}_i\ \mathbf{r}, sv; I'$ when $R(r) = \texttt{inj\_union}^i_{(c_1,c_2)}\ \ell_1$ and $H(\ell_1) = \langle \texttt{tag}_i, w\rangle$	$(H, R\{\mathbf{r} \mapsto \texttt{inj\_union}^i_{(\texttt{nil},[\times[\texttt{Tag}(\bar{i}),\tau_w]])}\ \ell_1\}, I_d)$ where $\hat{R}(sv) = \ell$ and $H(\ell) = \texttt{code}[]\Gamma.I_d$
$\texttt{brtgd}_i\ \mathbf{r}, sv; I'$ when $R(r) = \texttt{inj\_union}^i_{(c_1,c_2)}\ \texttt{tag}_k$	$(H, R\{\mathbf{r} \mapsto \texttt{inj\_union}^i_{(c_1,c_2')}\ \texttt{tag}_k\}, I_d)$ where c_2 normalizes to $\tau_0 :: \cdots :: \tau_k :: \cdots \tau_n$ and $\tau_k = \times[\texttt{Tag}(\bar{i}), \tau_w]$ and $c_2' = \tau_0 :: \cdots :: \tau_{k-1} :: \tau_{k+1} :: \cdots \tau_n$
$\texttt{brtgd}_i\ \mathbf{r}, sv; I'$ when $R(r) = \texttt{inj\_union}^i_{(c_1,c_2)}\ \ell$ and $H(\ell_1) = \langle \texttt{tag}_k, w\rangle\ (i \neq k)$	$(H, R\{\mathbf{r} \mapsto \texttt{inj\_union}^i_{(c_1,c_2')}\ \ell\}, I_d)$ where c_2 normalizes to $\tau_0 :: \cdots :: \tau_k :: \cdots \tau_n$ and $\tau_k = \times[\texttt{Tag}(\bar{i}), \tau_w]$ and $c_2' = \tau_0 :: \cdots :: \tau_{k-1} :: \tau_{k+1} :: \cdots \tau_n$
$\texttt{ret}$	$(H, R\{\mathbf{sp} \mapsto s'\}, I[\vec{c}/\Delta])$ where $R(\mathbf{sp}) = \ell[\vec{c}] \triangleright_{32} s'$ and $H(\ell) = \texttt{code}[\Delta]\Gamma.I$
$\texttt{jmp}\ sv$	$(H, R, I[\vec{c}/\Delta])$ where $\hat{R}(sv) = \ell[\vec{c}]$ and $H(\ell) = \texttt{code}[\Delta]\Gamma.I$

$(H, R, I) \longmapsto P$ where	
$\mathtt{unpack}[\alpha, \mathbf{r}], sv; I'$	$(H, R\{\mathbf{r} \mapsto w\}, I'[c/\alpha])$ where $\hat{R}(sv) = \mathtt{pack}\, w\, \mathtt{as}\, \tau\, \mathtt{hiding}\, c$
$\mathtt{vcase}(c, \alpha_1 . \mathtt{dead}\, sv, \alpha_2); I'$	$(H, R, I'[c'/\alpha_2])$ where c normalizes to $\mathtt{inj}_2\, c'$
$\mathtt{vcase}(c, \alpha_1, \alpha_2 . \mathtt{dead}\, sv); I'$	$(H, R, I'[c'/\alpha_1])$ where c normalizes to $\mathtt{inj}_1\, c'$
$\langle \beta, \gamma \rangle = c; I'$	$(H, R, I'[c_1, c_2/\beta, \gamma])$ where c normalizes to $\langle c_1, c_2 \rangle$
$(\mathtt{fold}\, \beta) = c; I'$	$(H, R, I'[c'/\beta])$ where c normalizes to $\mathtt{fold}_{\mu j.\kappa}\, c'$
$\mathtt{sub}_\tau\, \mathbf{r}, sv_1, sv_2; I'$	$(H, R\{\mathbf{r} \mapsto w_i\}, I')$ where $\hat{R}(sv_1) = \ell$ and $\hat{R}(sv_2) = i$ and $H(\ell) = [w_0, \ldots, w_{n-1}]$ and $0 \leq i < n$
$\mathtt{upd}_\tau\, sv_1, sv_2, sv_3; I'$	$(H\{\ell \mapsto [w_0, \ldots, w_{i-1}, \hat{R}(sv_3), w_{i+1}, \ldots, w_{n-1}]\}, R, I')$ where $\hat{R}(sv_1) = \ell$ and $\hat{R}(sv_2) = i$ and $H(\ell) = [w_0, \ldots, w_{n-1}]$ and $0 \leq i < n$
$\mathtt{fmov}\, \mathbf{f}, fv; I'$	$(H, R\{\mathbf{f} \mapsto \hat{R}(fv)\})$
$\mathtt{floadr}\, \mathbf{f}, \mathbf{r}; I'$	$(H, R\{\mathbf{f} \mapsto l\}, I')$ where $R(\mathbf{r}) = \ell$ and $H(\ell) = l$
$\mathtt{fstore}\, \mathbf{r}, fv; I'$	$(H\{\ell \mapsto l\}, R\{\mathbf{r} \mapsto \ell\}, I')$ where $\hat{R}(fv) = l$ and $\ell \notin H$
$\mathtt{fswrite}\, \mathbf{sp}(i), fv; I'$	$(H, R\{\mathbf{sp} \mapsto s'\}, I')$ where $s' = (R(\mathbf{sp}))[i]_{64} \leftarrow \hat{R}(fv)$
$\mathtt{sub}_\phi\, \mathbf{f}, sv_1, sv_2; I'$	$(H, R\{\mathbf{f} \mapsto l_i\}, I')$ where $\hat{R}(sv_1) = \ell$ and $\hat{R}(sv_2) = i$ and $H(\ell) = [l_0, \ldots, l_{n-1}]$ and $0 \leq i < n$
$\mathtt{upd}_\phi\, sv_1, sv_2, fv; I'$	$(H\{\ell \mapsto [l_0, \ldots, l_{i-1}, \hat{R}(fv), l_{i+1}, \ldots, l_{n-1}]\}, R, I')$ where $\hat{R}(sv_1) = \ell$ and $\hat{R}(sv_2) = i$ and $H(\ell) = [l_0, \ldots, l_{n-1}]$ and $0 \leq i < n$