# Certifying Compilation for Standard ML in a Type Analysis Framework.

**Leaf Eames Petersen**

May, 2005
CMU-CS-05-135

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis Committee:**
Robert Harper, Chair
Karl Crary
Peter Lee
Greg Morrisett (Harvard University)

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy*



This research was sponsored by the National Science Foundation under grant nos. CCR-0121633 and EIA-9706572, and the US Air Force under contract no. F-19628-95-C-0050. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

**Keywords:** Types, Compilers, Standard ML, Typed Assembly Language, Certifying Compilation, Typed Register Allocation, Type Analysis

**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.

In this thesis, I demonstrate that certified compilation is possible in a type analysis framework by extending the TILT/ML compiler to generate certified code in the form of typed assembly language without compromising the existing optimizations of the compiler. *This work demonstrates 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.*

*To my wife, Adriana*

# Acknowledgments

This thesis would not have come to be without the help of many people.

First among these are my mother and father, who first started me on the path of learning and provided me the freedom and the means to follow it as far as I desired. They gave me my education, in every sense.

All of the members of my committee provided excellent guidance and feedback, as well as many interesting conversations that informed this work. Bob Harper must be thanked especially: for teaching and mentoring me through this long process, for encouraging me to explore my ideas, and for seeing me through to the other side. Bob should also be thanked for relentlessly evicting the "wicked whiches" and other strange beasts that lurk in the dark corners of my prose. Karl Crary contributed a great deal to the framework underlying this work: that it was possible at all is a credit to the extent to which he foresaw the issues and prepared the ground. His insight and assistance were invaluable.

The system described in this thesis builds on the efforts of many other people. Without the work done on the **TILT** compiler by Perry Cheng, Chris Stone, and others, none of this would have been possible. I should especially like to thank David Swasey for doing a great deal of thankless work to make the **TALx86** backend (and the compiler in general) much more robust, usable, and correct. This thesis also benefited hugely from the excellent work done by Greg Morrisett and the rest of the **TALx86** team.

Many other people have offered me their help, support, and friendship while I have been at CMU, including Sharon Burks, Catherine Copetas, Frank Pfenning, Rose Hoberman, Derek Dreyer, Rowan Davies, Dan Spoonhower, Tara Cheesman, Chris Palmer, Carrie Sparks, and many others. Thanks for making my time in graduate school so much more enjoyable!

Leaf Petersen
May, 2005

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Types in compilation

It is an unfortunate fact about the state of programming that even good programs sometimes go wrong. It is not uncommon for programs to crash or misbehave, whether accidentally, 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 Java-script 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.

However, 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 produce undefined and unsafe behavior.

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 under compilation. 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 safety 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. Prior to this work however, type information was mostly erased in **TILT** well before the transformation to machine code was made, and hence safety properties of the resulting code could 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 have demonstrated 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. This dissertation gives a careful theoretical description of the key elements of the compilation process, and proves soundness theorems for the translations between the major intermediate languages. A description is also given of the actual implementation, including some empirical results.

It is becoming increasingly clear that type preserving compilation, in addition to serving as a mechanism for enabling safe mobile code, also provides great benefits in its own right as a compiler engineering technique. Just as type safe languages allow programmers to write correct code more quickly, type preserving compilation allows compiler implementers to write correct compilers more quickly. Type checking the intermediate representations of programs within the compiler allows many or most compiler bugs to be caught during compilation and to be localized to the particular point of failure in the compiler. Whole classes of pernicious compiler bugs (such as those that involve stack or memory corruption) can be eliminated by producing checkably type safe code. One of the results of this thesis is to provide further evidence of the efficacy of this technique, and to add to the body of experience in the engineering of type preserving compilers. This aspect of

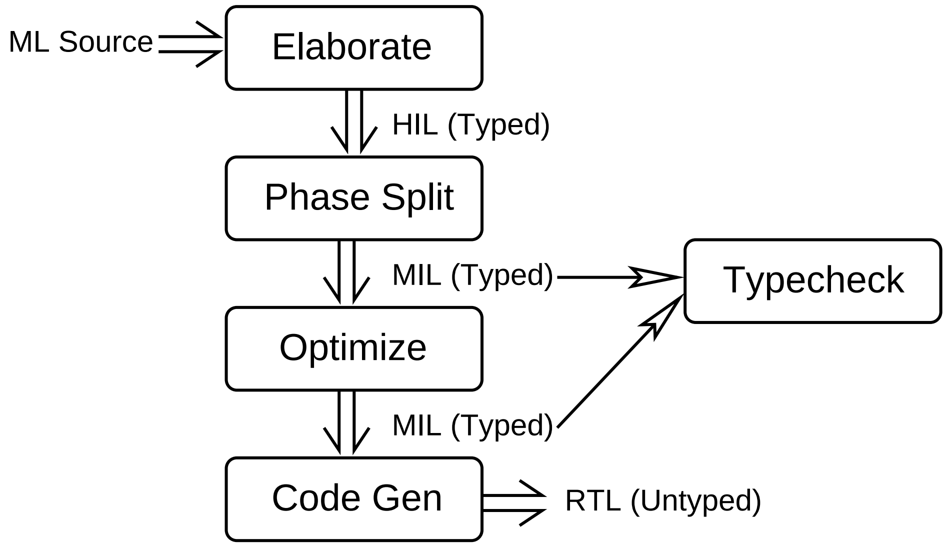



**Figure 1.1:** TILT Architecture

the implementation experience is discussed further in section 10.1.4.

## 1.2 The non-certifying TILT compiler

The past years have seen a great deal of interest in the idea of "typed compilers" that maintain type information deep into the compilation process. Such 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], and has been adopted by numerous other compilers, including the Glasgow Haskell Compiler. 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 1.1 depicts the structure of the non-certifying **TILT** compiler. Its 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 of 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 $\lambda_i^{ML}$ calculus. Nonetheless, the fact that some **MIL** primitives do indeed analyze their types mandates a type passing interpretation for the **MIL** operational semantics.

All of the intermediate languages of the **TILT** compiler up to and including the **MIL** are typed, and all of the compiler passes on these languages are type-preserving in the sense that they map well-typed programs to well-typed programs. Unfortunately for certification purposes, the subsequent languages from the **MIL** on down are not typed, and hence the generated code cannot in general be proven safe. This dissertation replaces this un-typed backend with a new type-preserving backend that produces certified code.

## 1.3 The certifying TILT compiler

One of the major goals of certifying compilation is to ensure that the certifying compiler is type-preserving: that is, that it maps well-typed programs in the source language to well-typed programs in the target language. In order to show that this is the case, I spend the first part of this dissertation presenting idealized versions of the compiler intermediate languages, and proving the soundness of the translations between them. The next two sections describe the idealized compiler and its relation to the implementation.

### 1.3.1 The theoretical compiler

The theoretical portion of this dissertation describes the framework for a translation mapping the original **TILT** internal language (the **MIL**, described in chapter 2) down to a typed assembly language that I call **TILTAL** (Typed Assembly Language for **TILT**). This translation uses as an intermediate stage a new internal language called the **LIL** (Low Level Language). The **LIL** is an impredicatively typed lambda calculus based on Crary and Weirich's LX [CW99]. Figure 1.2 describes the structure of the theoretical compiler.

The **LIL**, described in chapter 3, 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. 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. Chapter 4 gives a brief introduction to the type analysis methodology of the **LIL** via a worked example.

The translation from **MIL** to **LIL** serves to make type analysis and type representations explicit in the term language. This translation is described in detail in chapter 5. A subsequent pass mapping **LIL** terms to **LIL** terms performs closure conversion. The closure conversion translation is described in chapter 6.

Finally, I give a translation from the **LIL** into **TILTAL**. **TILTAL** 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

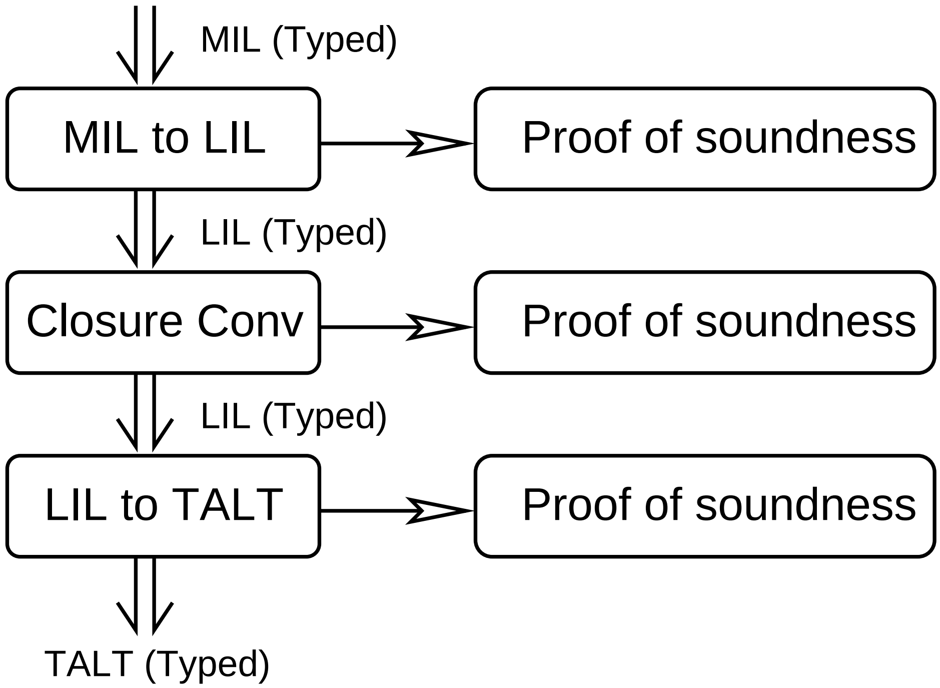



**Figure 1.2:** Structure of the theoretical compiler

collector) and array bounds checking. The **TILTAL** language is introduced in chapter 7, and the translation from **LIL** programs to **TILTAL** programs is given in chapter 8.

The theoretical **LIL** is actually very close to the **LIL** as implemented in the compiler, mostly lacking only an extended set of basic primitive operations. The presentation of the **TILTAL** 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 **TALx86** implementation departs significantly from that given here. However, for the most part the theoretical translations between the various languages are designed to capture all or most of the essential details of the implementation. In particular, the **LIL** to **TILTAL** translation of chapter 8 attempts as much as possible to account for the actual code generation techniques used by the implementation.

The main result of the theoretical portion of this dissertation is a proof that the compilation of **LIL** programs into **TILTAL** programs preserves types, in the sense that each of the individual translations is proved to map well-typed terms to well-typed terms. Proofs of soundness for the various translation phases are given in the chapters in which they are defined.

### 1.3.2 Compiler implementation

The second half of this thesis is an actual implementation of a certifying compiler for Standard ML. The theoretical compiler discussed in the previous section is intended as a model for the implementation. Where the original architecture (see figure 1.1) 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 1.3. The certifying **TILT** implementation is discussed in chapter 9.

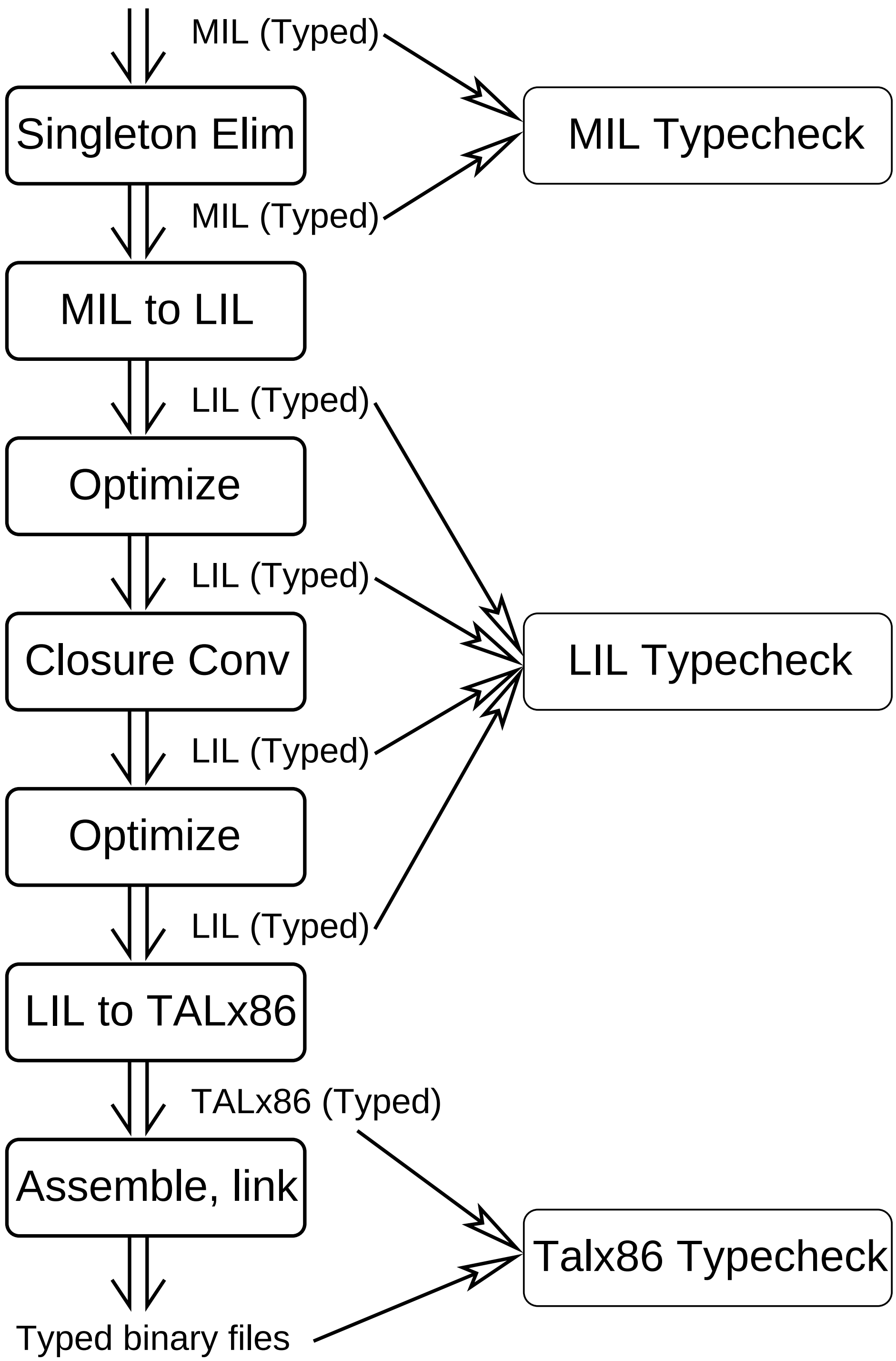



**Figure 1.3:** Structure of the certifying **TILT** backend

The implementation of certifying **TILT** required the addition of essentially five additional compiler stages.

- **MIL** singleton elimination
- Translation to **LIL**
- Closure conversion
- Optimization
- Translation to **TALx86**

To simplify the meta-theory of the **LIL**, I have chosen not to support the singleton kinds of the original **MIL** as implemented in the compiler. 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]. The implementation of this algorithm and its effects on compiled code are discussed in section 9.1.

The translation to **LIL** is described in detail in chapter 5. 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 **MIL** implements all of the optimizations that **TILT** supports, so extended optimization of **LIL** code is mostly unnecessary. However, the translations tend to produce code that can be improved significantly by a simple optimizer: for example, the closure converter frequently introduces dead code and projections from known records. Rather than making other phases do significantly more work to avoid this, it was simpler and cleaner to implement a basic optimization pass for the **LIL**. Note that the optimizer does not have a counterpart in the theoretical model.

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** must closure-convert types as data. 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** was implemented, closely modeled on the formal closure conversion translation from chapter 6.

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

Lastly, a translation is given mapping **LIL** programs into **TALx86** programs. Note that here, the implementation diverges significantly from the formal model described in chapter 8 in that the **TILTAL** and **TALx86** languages are quite different. However, the theoretical model was carefully designed to capture many of the interesting algorithmic approaches used in the actual implementation.

The final portion of the dissertation gives a basic quantitative evaluation of the implementation, including measurements of the size of the generated code and certificates to get a feeling for the overhead of certification. Some measurements of run time behavior of the compiled programs are also given.

It is important to state here that while I measure these quantities to gain some understanding of the cost of my approach, optimizing for small certificates and fast certification time is beyond the scope of this dissertation. 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, was not a primary goal.

## 1.4 Overview

In the next chapter, I provide a brief introduction to a simplified version of the **MIL** intermediate language which is the original source language for the compilation process described in this dissertation. Chapter 3 describes the **LIL** language which serves as the main intermediate language of the new backend. The complete static semantics of the **MIL** and the **LIL** are given in appendixes A and B, respectively. Chapter 4 gives an introduction to the style of type-analysis used in the **LIL** via an extended example. The translation from **MIL** to **LIL** is described in detail in chapter 5, and a proof of soundness is given. Closure conversion of **LIL** terms is described and proved sound in chapter 6. Chapter 7 introduces the **TILTAL** typed assembly language which serves as the target of the theoretical compiler. A translation from **LIL** programs to **TILTAL** programs is described and proved sound in chapter 8. Finally, the implementation and its relation to the theoretical presentation is discussed in chapter 9, and some empirical results about the implementation are presented.

# Chapter 2

# 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, VDP$^+$03]. 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 2.1. The **MIL** is a predicative lambda calculus based on Girard's $F_\omega$, extended with primitives for type analysis. The intention is that these primitives are definable in terms of the $\lambda_i^{ML}$ calculus of Harper and Morrisett [HM95]. In the untyped **TILT** back-ends, these primitives are only compiled directly after moving to an untyped calculus. The complete static semantics for the **MIL** are given in appendix A and are fairly straightforward. The remainder of this section gives a high-level overview of the structure of the language and describes the type-analysis methodology it employs.

## 2.1 Kinds and constructors

The kind structure of the **MIL** is relatively simple in the absence of singletons, with function and product kinds to classify constructor functions and tuples, and the base kind $\mathrm{T}_{32}$ to classify constructors which classify terms. The notation $\mathrm{T}_{32}$ indicates that the terms classified by constructors of this kind are intended to be represented by a 32 bit quantity after compilation. I use the term "constructor" in preference to the term "type" when refering to constructors of arbitrary or unspecified kind. I reserve the standard terminology "type" for constructors of kind $\mathrm{T}_{32}$.

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 in the tuple is small. In order to make this work with polymorphic functions, it is necessary to use type analysis.

$$
\begin{array}{lcl}
\kappa & ::= & \mathrm{T}_{32} \mid \kappa_1 \to \kappa_2 \mid \kappa_1 \times \kappa_2 \\
\\
c & ::= & \alpha \mid \texttt{Int} \mid \texttt{Boxf} \mid (\vec{c}) \to 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 & \texttt{Vararg}_{c_1 \to c_2} \mid \texttt{Vararg}_{c_1 \to^t c_2} \mid \texttt{Sum}_i(\vec{c}) \mid \texttt{Sum}_i^j(\vec{c}) \\
 & \mid & \texttt{Array}_c \mid \texttt{Farray} \mid \texttt{Exn} \mid \texttt{Dyntag}_c \\
\\
\tau & ::= & T(c) \mid \forall[\alpha\vec{::}\kappa](\vec{\tau})(i) \to \tau \mid \forall[\alpha\vec{::}\kappa](\vec{\tau})(i) \to^t \tau \mid \tau_1 \times \ldots \times \tau_n \\
\\
fv & ::= & \mathfrak{r} \mid x_f \\
\\
sv & ::= & x \mid i \mid \texttt{roll}_c\, sv \mid \texttt{unroll}_c\, sv \\
 & \mid & \texttt{inj\_tag}^j_{\texttt{Sum}_i(\vec{c})} \\
\\
opr & ::= & sv \mid \texttt{vararg}_{c_1 \to c_2}\, sv \mid \texttt{onearg}_{c_1 \to c_2}\, sv \\
 & \mid & \texttt{boxf}\, fv \mid \texttt{unboxf}\, sv \mid fv \\
 & \mid & \texttt{proj}^i\, sv \mid \texttt{inj}^j_{\texttt{Sum}_i(\vec{c})}\, sv \\
 & \mid & \langle \vec{sv} \rangle \mid \texttt{select}^i\, sv \\
 & \mid & \texttt{case}(sv)(x_1.e_1, \ldots, x_n.e_n) \mid \texttt{handle}_\tau(e_1, x.e_2) \\
 & \mid & sv[\vec{c}](\vec{sv})(\vec{fv}) \mid \texttt{inj\_dyn}_c(sv_1, sv_2) \\
 & \mid & \texttt{raise}_\tau\, sv \mid \texttt{mkexntag}_c \\
 & \mid & \texttt{exncase}_\tau(sv)(sv_1 \Rightarrow x_1.e_1, \_ \Rightarrow e_2) \\
 & \mid & \texttt{sub}_c(sv_1, sv_2) \mid \texttt{fsub}(sv_1, sv_2) \\
 & \mid & \texttt{upd}_c(sv_1, sv_2, sv_3) \mid \texttt{fupd}(sv_1, sv_2, fv) \\
 & \mid & \texttt{array}_c(sv_1, sv_2) \mid \texttt{farray}(sv, fv) \\
\\
e & ::= & sv \mid \texttt{let}_\tau\, \texttt{rec}_\tau\, f[\alpha\vec{::}\kappa](x\vec{:}\tau)(\vec{x_f}).e\, \texttt{in}\, e \\
 & \mid & \texttt{let}_\tau\, x = opr\, \texttt{in}\, e \mid \texttt{let}_\tau\, x_f = opr\, \texttt{in}\, e \\
\\
\Delta & ::= & \bullet \mid \Delta, \alpha{:}\kappa \\
\Gamma & ::= & \bullet \mid \Gamma, x{:}\tau \mid \Gamma, x_{64}
\end{array}
$$

**Figure 2.1: MIL** syntax

The `Vararg` type can be thought of as simply a type level typecase in a $\lambda_i^{ML}$ 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.

## 2.2 Proper types

Also described in figure 2.1 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 tuple type at the type level indicates the possibility of constructing pairs containing polymorphic functions which is not provided for by the constructor level.

## 2.3 Terms

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 [FSDF93, Mog89] to reflect the form used internally in the **TILT** compiler. Named form forces incremental calculations to be given names via variable binding (hence *named* form) which is important for various compiler passes and code generation. 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 $\lambda_i^{ML}$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 that 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 re-box 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 $\texttt{vararg}_{c_1 \to c_2}\, sv$ corresponds roughly to the following code using an explicit $\lambda_i^{ML}$ 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 chapter discusses a strategy for doing this by reflecting type analysis into the term level of a more powerful lambda calculus.

# Chapter 3

# 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 originally 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. This untyped approach was taken because of the absence of a well-understood type system for typing such code.

In order to make this ad-hoc solution viable for producing typed certified code, 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.

## 3.1 The LIL syntax and static semantics

Figure 3.1 describes the complete syntax of the **LIL**. As with the **MIL**, the language is syntactically restricted to named form. This is not particularly necessary for theoretical purposes, but matches more closely the implementation.

Kinds form the top of the syntactic hierarchy, and are generally written using the meta-variable $\kappa$. Kinds classify the constructors, for which the meta-variables $c$ is generally used. In addition the meta-variables $\tau$ and $\phi$ are used to distinguish constructors of kind $\mathrm{T}_{32}$, $\mathrm{T}_{64}$ respectively. This is purely a presentational distinction however and does not correspond to an actual syntactic distinction.

At the term level, the classes of small 32 bit and 64 values are notated as *sv* and *fv* respectively; 32 bit and 64 bit operations are notated as *opr* and *fopr* respectively; and expressions are notated using the meta-variable $e$. Note that the distinction between 32 and 64 bit values is a syntactic distinction and not merely a notational convention, and similarly for operations.

**LIL** programs ($p$) consist of a mutually recursive set ($d$) of heap bindings (*hval*), and an executable expression $e$. In the theoretical presentation here, code functions are the only heap values

$$
\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, \ldots, \kappa_n] \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}_i^{+[k_i]}\, c \mid \mathtt{case}(c, [\alpha_1.c_1, \ldots, \alpha_n.c_n]) \\
 & & \mid \mathtt{fold}_{\mu j.k}\, c \mid \mathtt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}\,\mathtt{in}\, c) \\
 & & \mid \mathtt{Float} \mid \mathtt{Int} \mid \mathtt{Boxed} \mid \mathtt{Void} \mid \times \mid \to \mid \mathtt{Code} \\
 & & \mid c[\kappa] \mid \Lambda j.c \mid \exists \mid \forall \mid \mathtt{Rec} \mid \bigvee \\
 & & \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 \ell \mid i \\
 & & \mid \mathtt{inj\_union}_c\, sv \mid \mathtt{inj\_dyn}_\tau(sv_1, sv_2) \\
 & & \mid \mathtt{unroll}_\tau\, sv \mid \mathtt{roll}_\tau\, sv \\
 & & \mid \mathtt{pack}\, sv\, \mathtt{as}\, \exists\alpha{:}\kappa.\tau\ \mathtt{hiding}\, c \mid sv[c] \mid \mathtt{tag}_i \\
 & & \\
fopr & ::= & fv \mid \mathtt{unbox}\, sv \mid \mathtt{sub}_\phi(sv_1, sv_2) \\
opr & ::= & sv \mid \mathtt{select}^i\, sv \\
 & & \mid \mathtt{case}(sv)(x.e_1, \ldots, x.e_n) \\
 & & \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{box}\, fv \mid \langle \vec{sv} \rangle \\
 & & \mid sv(\vec{sv})(\vec{fv}) \mid \mathtt{call}\, sv(\vec{sv})(\vec{fv}) \\
 & & \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[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](x_1{:}\tau_1, \ldots, x_m{:}\tau_m)(z_1{:}\phi_1, \ldots, z_k{:}\phi_k).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 \\
 & & \mid \mathtt{let}\ (\mathtt{inj}_i\, \beta) = (c, sv)\ \mathtt{in}\, e \\
 & & \\
hval & ::= & \mathtt{code}_\tau[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](x_1{:}\tau_1, \ldots, x_m{:}\tau_m)(z_1{:}\phi_1, \ldots, z_k{:}\phi_k).e \\
d & ::= & \epsilon \mid d, \ell{:}\tau \mapsto hval \\
p & ::= & \mathtt{letrec}\, d\ \mathtt{in}\, e \\
 & & \\
\Psi & ::= & \bullet \mid \Psi, \ell{:}\tau \\
\Delta & ::= & \bullet \mid \Delta, j \mid \Delta, \alpha{:}\kappa \\
\Gamma & ::= & \bullet \mid \Gamma, x{:}\tau \mid \Gamma, x_{64}{:}\phi
\end{array}
$$

**Figure 3.1: LIL** syntax

permitted: however, in practice additional statically allocated data is placed in the heap section.[1]

The next several sections discuss some of the interesting syntactic aspects of the **LIL** and introduce the relevant typing judgements. The complete static semantics for the **LIL** can be found in appendix B. Many of the constructs described here are also discussed in detail by Crary and Weirich [CW99].

### 3.1.1 Typing contexts

| Context judgements | |
|---|---|
| Heap contexts | $\vdash \Psi$ **ok** |
| Constructor contexts | $\vdash \Delta$ **ok** |
| Term contexts | $\Delta \vdash \Gamma$ **ok** |

There are three sorts of typing contexts for the **LIL**: heap contexts $\Psi$, constructor contexts $\Delta$, and term contexts $\Gamma$. Heap contexts associate closed types with labels and are derived from the top level heap of a program. Constructor contexts actually serve two purposes: they bind free kind variables, and they bind free constructor variables at kinds (which may refer to previously bound kind variables). In principle these two contexts could be separated, but since kind variables are uni-typed it seems unnecessary. Term contexts bind 32 and 64 bit term variables at types, which may refer to previously bound constructor variables. I assume that variables for each of the different syntactic classes (constructors, kinds, 32 bit terms and 64 bit terms) are drawn from mutually disjoint infinite sets.

A heap context is well-formed if all of the labels in its domain are distinct, and if each of the types in its range is well-formed in an empty constructor context. This latter constraint reflects the fact that heap values are bound at the top-level of programs.

A constructor context is well-formed if all of the kind variables in its domain are distinct; and if all of the constructor variables in its domain are distinct, and if each type in its range is well-formed in the context preceding its binding.

A term context is well-formed if all of the 32 bit variables in its range are distinct, each type at which a 32 bit variable is bound is well-formed at kind $\mathrm{T}_{32}$; and if all of the 64 bit variables in its range are distinct, and if each type at which a 64 bit variable is bound is well-formed at kind $\mathrm{T}_{64}$.

### 3.1.2 Constructors and Kinds

| Constructor and kind judgements | |
|---|---|
| Kinds | $\Delta \vdash \kappa$ **ok** |
| Constructors | $\Delta \vdash c : \kappa$ |
| Equivalence | $\Delta \vdash c \equiv c' : \kappa$ |

The constructor and kind typing judgements are unsurprising. The judgement $\Delta \vdash \kappa$ **ok** defines what it means for a kind $\kappa$ to be well-formed in a constructor context $\Delta$. The judgement $\Delta \vdash c : \kappa$

[1]The practice of referring to the static data segment of **LIL** programs as a "heap" reflects the standard terminology of the literature in this area: however, this usage is slightly misleading since in practice a **LIL** "heap" is understood to correspond more closely to the code and data segments of an executable image, rather than to dynamically allocated heap space. Nonetheless, since this distinction is not apparent in the dynamic semantics of **TILTAL**, and since the usage has become standard in the literature, I will continue to refer to this portion of a **LIL** program as the heap. Sorry Bob!

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

**Figure 3.2:** Constructor constants and their kinds

defines what it means for a constructor $c$ to be well-formed at a kind $\kappa$ in a context $\Delta$. And the judgement $\Delta \vdash c \equiv c' : \kappa$ defines when two constructors are equivalent. Note that the kind and the context in the equivalence judgement are present for technical reasons: they should not change the set of equivalences on well-formed constructors (though they do rule out equivalences on ill-formed constructors). The complete definitions of these judgements are given in appendix B.

The most important change from the **MIL** is the enrichment of the kind structure: most importantly, 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. In order to provide for the possibility of more general 64 bit types, the **LIL** uses an explicit kind distinction, providing kinds $\mathrm{T}_{32}$ and $\mathrm{T}_{64}$ corresponding to the kind of the types of 32 and 64 bit expressions respectively.

The meta-variables $c$, $\tau$ and $\phi$ are used to represent constructors of arbitrary kind, kind $\mathrm{T}_{32}$ and kind $\mathrm{T}_{64}$ 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, $\texttt{fold}_{\mu j.\kappa}\, c$, 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 $\texttt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa').c)$ 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 $\alpha$ 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 sub-component of the argument (and hence is guaranteed to terminate), $\alpha$ is bound at kind $\kappa$ with occurrences of $j$ left abstract, and the recursive variable $\rho$ always has $j$ as its domain kind.

The constructors classifying the term level are presented in the **LIL** as constants of higher kind. These constructors, given in figure 3.2, 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 dissertation I will frequently use the usual ML list syntax for constructor lists. The kind $\texttt{nat}$ 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 $\rightarrow$ constructor correspond to the types of the 32 bit and 64 bit arguments of the function, and its return type. Hence, $\rightarrow([\tau])([\phi])(\tau)$ corresponds to $(\tau;\phi) \rightarrow \tau$.

This higher-order abstract syntax methodology is very convenient for both the theory and the implementation but can interfere with readability. For example: the type of the polymorphic pairing function written in this style appears as

$$\forall[\mathrm{T}_{32}](\lambda(\alpha{:}\mathrm{T}_{32}).\forall[\mathrm{T}_{32}](\lambda(\beta{:}\mathrm{T}_{32}). \rightarrow [a][](\rightarrow[b][](\times[a,b]))))$$

(Note that even here I have used derived notation for lists). To enhance readability, I will frequently make use of more standard notation. So in this example, I would write the type as:

$$\forall[\alpha{:}\mathrm{T}_{32}, \beta{:}\mathrm{T}_{32}].\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$$

This practice is also done extensively in the actual implementation via libraries which provide defined forms implementing the more standard type constructors in terms of the HOAS style constructors.

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.

Sum types are dealt with in the **LIL** using union types. In principle we allow injection into any union type. However, the case construct limits its arguments to have union types composed solely of tags or tagged records for which the tags are disjoint, and cover the full range of tags from the zero to the largest tag. So for example, the type $\bigvee[\texttt{Tag}(0), \times[\texttt{Tag}(1), \tau]]$ is inhabited by terms which are either $\texttt{tag}_0$ or a pair containing a $\tau$ in its second field, tagged with $\texttt{tag}_1$ in the first field. This union type therefore describes a valid argument to case. In point of fact, the tag on the pair is not necessary and could be elided. More generally, when $\tau$ is a pointer, then both the tag and the pointer indirection can be eliminated. Note though that in the presence of unknown types such an optimization again requires type analysis. The actual implementation of sums in **TILT** in fact performs this optimization. However, since it adds nothing new to the problem, the theoretical discussion here assumes a simple treatment of sums in which all value-carrying arms are tagged.

The universal and existential constructors take as arguments a kind indicating the domain of quantification 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. $\texttt{Rec}[\kappa](\lambda(\rho{:}\kappa{\rightarrow}\mathrm{T}_{32}).\lambda(\alpha{:}\kappa).c)(c')$ (where $c'$ has kind $\kappa$) corresponds to recursively defining a constructor of kind $\kappa{\rightarrow}\mathrm{T}_{32}$ and applying it to $c'$. Within the body of $c$, $\rho$ stands for the recursively defined constructor, and $\alpha$ 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 $c_r = (\lambda(\rho{:}\kappa \rightarrow \mathrm{T}_{32}).\lambda(\alpha{:}\kappa).c)$.

In order to allow closure conversion to be done within the **LIL** language, two "function" types are given. The $\rightarrow$ primitive type constructor when applied to arguments classifies functions in the usual sense. The `Code` primitive on the other hand, classifies "code" functions: that is, code that is closed.

### 3.1.3 Terms

| Term judgements | |
|---|---|
| Small values | $\Psi; \Delta; \Gamma \vdash sv : \tau$ |
| 64 bit values | $\Psi; \Delta; \Gamma \vdash fv : \phi$ |
| 32 bit Operations | $\Psi; \Delta; \Gamma \vdash opr : \tau\ \mathbf{opr}_{32}$ |
| 64 bit Operations | $\Psi; \Delta; \Gamma \vdash fopr : \phi\ \mathbf{opr}_{64}$ |
| Expressions | $\Psi; \Delta; \Gamma \vdash e : \tau\ \mathbf{exp}$ |
| Heap values | $\Psi \vdash h : \tau\ \mathbf{hval}$ |
| Heaps | $\Psi \vdash d\ \mathbf{ok}$ |
| Programs | $\Psi \vdash p : \tau$ |

The expression level of the **LIL** is divided into five syntactic classes: 64 bit values ($fv$), 32 bit values ($sv$), 32 bit operations ($opr$), 64 bit operations ($fopr$) and expressions ($e$). Programs are syntactically restricted to a named form where all intermediate computations are named in the usual fashion. In addition to the five classes making up expressions, there are an additional three syntactic classes making up full **LIL** programs: heap values ($hval$), heaps ($d$), and programs ($p$).

The well-formedness judgements for the five expression classes are all of the same essential form, defining what it means for a term to be well-formed at a type $\tau$ (or $\phi$) in heap context $\Psi$, constructor context $\Delta$, and term context $\Gamma$. The other three judgements apply to heaps and programs which do not have free variables of any sort. Therefore, the only context present in these judgements is the heap context, which describes the free labels of a heap or a program. As usual, any heap value may refer to the label bound to any other heap value: that is, all the heap values are mutually recursively defined. The presence of a heap context in the program well-formedness judgement leaves open the possibility of compiling programs against externally defined labels.

#### Small values

Values include variables, constants, polymorphic instantiation, and existential introduction. Sum tags $\texttt{tag}_i$ are made explicit, and a coercion is introduced to inject terms into the union type.

#### 64 bit values

The only 64 bit values present in the **LIL** are 64 bit float constants and variables.

#### 32 bit operations

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 introduction and tuple selection, boxing, sum case, known sum projection, exception constructs, and boxing of 64 bit values. Array update and creation operations are provided for both 32 and 64 bit arrays, along with the array subscript operation for 32 bit arrays. Note that all memory allocation in the **LIL** is explicit and present in this level, whether through the tuple introduction, the box primitive or the array creation primitives. The only exception to this is function introduction which implicitly allocates a closure: this is dealt with via closure conversion which turns uses of functions into uses of tuples and code. This is described in detail in chapter 6 and section 9.3.

### 64 bit operations

The 64 bit operations include unboxing of 64 bit values and 64 bit array subscripts, as well as the inclusion of 64 bit values.

### Expressions

Expressions in the **LIL** are either small values, or let bindings of any of several forms. Recursive function binding, 32 bit variable binding, and 64 bit variable binding are provided. Existential unpacking is also included at this level. The most interesting expressions however, are the type refinement bindings that support the technology upon which type analysis in the **LIL** is built.

The most important of these 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 $\alpha$ gets instantiated with a left or a right injection. But since there are no closed values of type `Void`, it is apparent that $\alpha$ 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 $\alpha$ 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 $e$, $\alpha$ 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.

In addition, two special binding forms exist for refining constructor paths into variables for analysis. When $\alpha$ has kind $\kappa_1 \times \kappa_2$, the pair refinement operator $\mathtt{let}\langle\beta, \gamma\rangle = \alpha\, \mathtt{in}\, e$ replaces all occurrences of $\alpha$ in e with the pair $\langle\beta, \gamma\rangle$. This means that projections from $\alpha$ can be turned into variables. The `vcase` construct can only refine the type of its argument when the argument is a variable: this pair refinement 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 $\alpha$ has a recursive kind.

### Heap values

The only heap values currently supported in the theoretical treatment of the **LIL** are code functions, which are necessary for closure conversion. Each heap value is required to be closed with respect to variables, but may refer to any other heap value via its label.

**Heaps**

A heap is a collection of mutually recursively defined heap values. A heap is well-formed in a heap context if each of its constituent heap values is well-formed at the type at which its label is bound. Note that I require that all of the heap labels be present in the heap context: for non-recursive functions this is not strictly necessary, but it serves the additional purpose of enforcing the property that all labels in the heap are distinct (via well-formedness check on heap contexts).

$$\frac{\vdash \Psi\ \mathbf{ok}}{\Psi \vdash \epsilon\ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\Psi[\ell{:}\tau] \vdash hval : \tau\ \mathbf{hval}\\ \Psi[\ell{:}\tau] \vdash d\ \mathbf{ok}\end{array}}{\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval\ \mathbf{ok}}$$

**Programs**

**LIL** programs consist of a heap, which binds labels to heap values, and an expression which computes the "value" of the program. Since the heap values are (potentially) mutually recursive, the heap is checked in a context including bindings for all of the labels in the heap. For notational purposes, I define an operation on heaps, $\Psi(d)$, that corresponds to the heap context produced by taking the label and type from each binding in a heap

$$\begin{array}{ll}\Psi(\epsilon) & \stackrel{\text{def}}{=} \bullet\\ \Psi(d, \ell{:}\tau \mapsto hval) & \stackrel{\text{def}}{=} \Psi(d), \ell{:}\tau\end{array}$$

The well-formedness rule for program simply checks the heap and the expression portion of the program in the heap context extended with the bindings from the heap.

$$\frac{\begin{array}{c}\Psi, \Psi(d) \vdash \quad \mathbf{ok}\Psi, \Psi(d) \vdash d\ \mathbf{ok}\\ \Psi, \Psi(d); \bullet; \bullet \vdash e : \tau\ \mathbf{exp}\end{array}}{\Psi \vdash \mathtt{letrec}\, d\, \mathtt{in}\, e : \tau}$$

## 3.2 Useful properties of the LIL

Subsequent proofs rely on certain properties of the **LIL**: in particular, that well-typedness is preserved under substitution and that weakening for typing contexts is admissible.

**Lemma 1 (LIL Substitution)**
*If* $\Delta, \alpha{:}\kappa' \vdash c : \kappa$ *and* $\Delta \vdash c' : \kappa'$ *then* $\Delta \vdash c[c'/\alpha] : \kappa$.

**Proof:** (By induction on c)
We proceed by cases on the last rule of the derivation.

1. If c is a constant or $*$, then $c[c'/\alpha] = c$. By assumption, $\Delta, \alpha{:}\kappa_1 \vdash c : \kappa$, so by strengthening, $\Delta \vdash c : \kappa$.

2. $\Delta[\alpha{:}\kappa'] \vdash \alpha' : \kappa$

   - $\alpha' = \alpha$. Then $c[c'/\alpha] = c_1$. By assumption, $\Delta \vdash c' : \kappa'$ and $\Delta, \alpha{:}\kappa' \vdash \alpha : \kappa$. But by inspection of the typing rules, it is clear that the derivation must end in a use of the variable rule, so $\kappa = \kappa'$, and hence $\Delta \vdash c' : \kappa$
   - $\alpha' \neq \alpha$. Then the result follows as in the constant case above.

3. $\Delta[\alpha{:}\kappa'] \vdash \lambda(\beta{:}\kappa_\beta).c : \kappa_\beta \to \kappa_r$. Then by assumption, $\Delta, \alpha{:}\kappa', \beta{:}\kappa_\beta \vdash c : \kappa_r$. By induction, $\Delta, \beta{:}\kappa_\beta \vdash c[c'/\alpha] : \kappa_r$. (Note that we assume that $\beta$ is chosen appropriately to avoid capture.) Then by the lambda rule, $\Delta \vdash \lambda(\beta{:}\kappa_\beta)(c[c'/\alpha]) : \kappa$, and hence by definition of substitution, $\Delta \vdash (\lambda(\beta{:}\kappa_\beta)c)[c'/\alpha] : \kappa$.

4. $\Delta[\alpha{:}\kappa'] \vdash c_1\ c_2 : \kappa$. Then by assumption, $\Delta, \alpha{:}\kappa' \vdash c_1 : \kappa_2 \to \kappa$ and $\Delta, \alpha{:}\kappa' \vdash c_2 : \kappa_2$. By induction, $\Delta \vdash c_1[c'/\alpha] : \kappa_2 \to \kappa$ and $\Delta \vdash c_1[c'/\alpha] : \kappa_2$. The result follows directly then from the application rule and the definition of substitution: $\Delta \vdash (c_1\ c_2)[c'/\alpha] : \kappa$.

5. $\Delta[\alpha{:}\kappa'] \vdash \langle c_1, c_2 \rangle : \kappa_1 \times \kappa_2$. Then by assumption $\Delta, \alpha{:}\kappa' \vdash c_1 : \kappa_1$ and $\Delta, \alpha{:}\kappa' \vdash c_2 : \kappa_2$. By induction, $\Delta \vdash c_1[c'/\alpha] : \kappa_1$ and $\Delta \vdash c_2[c'/\alpha] : \kappa_2$. The result follows directly then from the pair rule and the definition of substitution: $\Delta \vdash \langle c_1, c_2 \rangle[c'/\alpha] : \kappa$.

6. $\Delta[\alpha{:}\kappa'] \vdash \pi_1 c_p : \kappa_1$. Then by assumption $\Delta, \alpha{:}\kappa' \vdash c_p : \kappa \times \kappa_2$. By induction, $\Delta \vdash c_p[c'/\alpha] : \kappa \times \kappa_2$. The result follows from the the projection rule and the definition of substitution.

7. $\Delta[\alpha{:}\kappa'] \vdash \pi_2 c_p : \kappa_2$. As for $\pi_2$.

8. $\Delta[\alpha{:}\kappa'] \vdash \mathtt{inj}_i\, c : +[\kappa_1, \ldots, \kappa_n]$. Then by assumption $\Delta, \alpha{:}\kappa' \vdash c : \kappa_i$. By induction, $\Delta \vdash c[c'/\alpha] : \kappa_i$. Finally, by the injection rule and the definition of substitution:

$$\Delta, \alpha{:}\kappa' \vdash (\mathtt{inj}_i c)[c'/\alpha] : +[\kappa_1, \ldots, \kappa_n]$$

9. $\Delta[\alpha{:}\kappa'] \vdash \mathtt{case}\, c[(\alpha_1.c_1, \ldots, \alpha_n.c_n)] : \kappa$. By assumption:

   - $\Delta, \alpha{:}\kappa' \vdash c : +[\kappa_1, \ldots, \kappa_n]$
   - $\Delta, \alpha{:}\kappa', \alpha_i{:}\kappa_i \vdash c_i : \kappa$

   By induction:

   - $\Delta \vdash c[c'/\alpha] : +[\kappa_1, \ldots, \kappa_n]$
   - $\Delta[\alpha_i{:}\kappa_i] \vdash c_i[c'/\alpha'] : \kappa$

   (Note that the $\alpha_i$ may always be chosen to avoid capture). Then by the case rule and the definition of substitution, $\Delta \vdash (\mathtt{case}\, c[(\alpha_1.c_1, \ldots, \alpha_n.c_n)])[c'/\alpha] : \kappa$

10. $\Delta[\alpha{:}\kappa'] \vdash \mathtt{fold}_{\mu j.\kappa}\, c : \mu j.\kappa$. Then by assumption, $\Delta[\alpha{:}\kappa'] \vdash c : \kappa[\mu j.\kappa/j]$ and $\Delta[\alpha{:}\kappa'] \vdash \mu j.\kappa\ \mathbf{ok}$. By induction, $\Delta[\alpha{:}\kappa'] \vdash c[c'/\alpha] : \kappa[\mu j.\kappa/j]$, and so by the fold rule and the definition of substitution, $\Delta[\alpha{:}\kappa'] \vdash (\mathtt{fold}_{\mu j.\kappa}\, c)[c'/\alpha] : \kappa[\mu j.\kappa/j]$

11. $\Delta[\alpha{:}\kappa'] \vdash \mathtt{pr}(j, \beta{:}\kappa_1, \rho{:}(j \to \kappa_2),\ \mathtt{in}\, c) : \mu j.\kappa_1 \to \kappa_2$ Then by assumption,

$$\Delta[\alpha{:}\kappa'] \vdash \mathtt{pr}(j, \beta{:}\kappa_1, \rho{:}(j \to \kappa_2),\ \mathtt{in}\, c) : \mu j.\kappa_1 \to \kappa_2$$

By induction

$$\Delta, j, \beta{:}\kappa_1, \rho{:}(j \to \kappa_2) \vdash c[c'/\alpha] : \kappa_2$$

Note that we can choose variables appropriately to avoid capture, and hence by the primitive recursion rule and the definition of substitution

$$\Delta[\alpha{:}\kappa'] \vdash \mathtt{pr}(j, \beta{:}\kappa_1, \rho{:}(j \to \kappa_2),\ \mathtt{in}\, c)[c'/\alpha] : \mu j.\kappa_1 \to \kappa_2$$

12. $\Delta[\alpha{:}\kappa'] \vdash c[\kappa] : \kappa_2[\kappa/j]$.

Then by assumption:
$\Delta[\alpha{:}\kappa'] \vdash c : \forall j.\kappa_2$

By induction:
$\Delta \vdash c[c'\alpha] : \forall j.\kappa_2$

By the kind application rule and the definition of substitution:
$\Delta \vdash c[\kappa][c'/\alpha] : \kappa_2[\kappa/j]$

13. $\Delta[\alpha{:}\kappa'] \vdash \Lambda j.c : \forall j.\kappa$. Then by assumption, $\Delta[\alpha{:}\kappa'], j \vdash c : \kappa$. By induction $\Delta, j \vdash c[c'/\alpha] : \kappa$. Hence by the kind abstraction rule and the definition of substitution $\Delta \vdash (\Lambda j.c)[c'/\alpha] : \kappa$.

∎

A few structural properties of **LIL** contexts are important as well. I state the weakening property and give a sketch of the proof. I remain informal about re-ordering properties of typing contexts throughout, but in the absence of dependent types or kinds it is straightforward (though tedious) to formalize them. I will also occasionally informally use simultaneous weakening to combine whole contexts: e.g. $\Delta, \Delta'$ where the intersection of the domains is empty. It should be clear that this form of weakening can also be formalized by an induction on the second context, appealing to the unary forms of the weakening lemmas to incrementally construct the new context.

**Lemma 2 (LIL Weakening)**

1. *If* $\Delta \vdash \kappa$ **ok** *and* $j \notin \Delta$, *then* $\Delta, j \vdash \kappa$ **ok**.
2. *If* $\Delta \vdash \kappa$ **ok** *and* $\Delta \vdash \kappa'$ **ok** *and* $\alpha \notin \Delta$, *then* $\Delta, \alpha{:}\kappa' \vdash \kappa$ **ok**.
3. *If* $\Delta \vdash c : \kappa$ *and* $\Delta \vdash \kappa'$ **ok** *and* $\alpha \notin \Delta$ *then* $\Delta, \alpha{:}\kappa' \vdash c : \kappa$.
4. *If* $\Delta \vdash c \equiv c' : \kappa$ *and* $\Delta \vdash \kappa'$ **ok** *and* $\alpha \notin \Delta$ *then* $\Delta, \alpha{:}\kappa' \vdash c \equiv c' : \kappa$.

**Proof (Sketch):** Each of the proofs proceeds similarly by induction on the structure of typing derivations. For each typing rule, I inductively construct new sub-derivations for each premise and use the original rule to construct the new derivation. The side-condition on binding sites does not follow immediately: it is necessary to observe that it is always possible to use alpha-variance to choose an appropriate bound variable different from $\alpha$. For premises of the form $\vdash \Delta$ **ok**, note that the derivation of $\vdash \Delta, \alpha{:}\kappa'$ **ok** follows directly from the assumptions and the definition of context well-formedness.

∎

I also state an inversion property of **LIL** operations.

**Lemma 3 (LIL inversion)**

1. *If $D$ is a derivation of $\Psi; \Delta; \Gamma \vdash opr : \tau$ $\mathbf{opr}_{32}$, then $D$ has a unique last rule for any choice of $opr$.*

2. *If $D$ is a derivation of $\Psi; \Delta; \Gamma \vdash fopr : \phi$ $\mathbf{opr}_{64}$, then $D$ has a unique last rule for any choice of $opr$.*

**Proof:** By inspection. No two operation rules apply to the same syntactic construct. ∎

# Chapter 4

# Example: Floating point array flattening

## 4.1 An optimized array strategy

The use of the **LIL** type refinement operations discussed in the previous chapter is somewhat non-intuitive. Before attempting to give a full account of the translation of the **MIL**, it is useful to consider a small example demonstrating the type-analysis methodology used 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 $\lambda_i^{ML}$ 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.

## 4.2 LIL implementation of optimized arrays

Type analysis in the **LIL** is based on the idea of first encoding types as abstract syntax trees, and then writing functions that use the encodings to reconstruct the actual types or to choose different branches of code. In fact, it is not necessary to encode entire types: we may choose an encoding that captures only the information that is useful for our purposes.

More concretely, we first choose an encoding strategy that captures the features of interest about a type. For every type of interest (that is, every type which is to be analyzed) we construct two object level items: a constructor which serves as a *static encoding* of the type (**SE**), and a term which serves as a *dynamic encoding* of the type (**DE**). Static encodings of types are used during typechecking to reconstruct the encoded type and to connect the type to its dynamic encoding. Dynamic encodings are used to perform the actual runtime dispatch. For each of these two encodings, there must also be a corresponding classifier describing it: the static encoding of a type is classified by the *static encoding kind* (**SEK**), and the dynamic encoding of a type is classified by its *dynamic encoding type* (**DET**).

### 4.2.1 Static encodings

At the kind level, sums and recursive kinds are used to build a type's static encoding. For the purposes of this example optimization, the encoding can be very simple: we need only to be able to distinguish between boxed 64-bit types and other types. This is reflected in the static encoding kind for our example:

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

The $\mathrm{T}_{\mathrm{opt}}$ kind classifies static encodings. 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 $\mathrm{T}_{\mathrm{opt}}$ encode types. In order to take advantage of these encodings, we replace all constructors by their encodings. For example, polymorphic functions must expect encoded types as their type arguments. In order to be able to typecheck applications of these functions, we must be able to reconstruct the original type from its encoding. This is done by writing an object level function to *interpret* static encodings into the types they encode. For the example at hand, this function simply boxes up the 64 bit types and leaves the 32 bit types unchanged. Hence:

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

The use of this interpretation function can be seen by considering the static encodings of boxed 64-bit floats and 32-bit ints:

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

Notice that the interpretation function maps each of these back to the original type. This allows us to reconstruct the original type from the encoding. An important point to note here is that this 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 or the other.

Of course, simply writing the interpretation is fairly uninteresting by itself since it merely gets us back to where we were before we encoded types. The real benefit of encoding types in this fashion comes from other useful functions that we can write using encodings of types. In particular, the actual type of specialized arrays that was given informally above can now be written down directly.

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

The $\mathtt{Array}_{\mathrm{opt}}$ constructor is a function that maps encoded types to optimized array types. It takes advantage of the additional information present in the encoded types to represent arrays of boxed floats more efficiently. When the encodings are statically known, the optimized array type can be reduced directly to an underlying primitive array type.

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

When the actual encoding is abstract (for example in a polymorphic function) the composite array constructor can not be reduced definitively to a primitive array type, since it is not statically known which sort of array can be expected.

### 4.2.2 Dynamic encodings

Static encodings account for type analysis at the type level by representing types as constructor level abstract syntax trees and by writing functions which analyze the structure of encodings to produce optimized results. Type analysis at the term level is accounted for in an analogous fashion using dynamic encodings. In addition to encoding every type at the constructor level, we also give a dynamic encoding of every type at the term level. The **DE** of a type can then be used to dispatch at runtime. A key point of this methodology is that the dynamic and static encodings of a type are not independent: they are related via the dynamic encoding type. This allows information gained via tests on the dynamic encoding to be reflect back onto the static encoding.

In fact, the **DET** of a type is simply another object level type function operating on the **SE** of the type in the same fashion as the interpretation function and the optimized array functions. For the example at hand, the type of the dynamic encoding of a type whose static encoding is $c$ is given by $R(c)$, where R is defined as follows:

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

This definition is a bit subtle, and is worth examining in detail. The principle high-level point is that dynamic encodings of types will be values of sum type, which can be dispatched on using the term level case construct. The subtlety arises in the types given for the two arms of the sum type here. Metaphorically speaking, each arm of the sum here can be thought of as serving as a “proof” of the identity of $\alpha$. The reasoning behind this is to observe that there are no closed values of type `Void`. This tells us that any closed value of type $\mathtt{case}\,c\;\mathtt{of}\;\mathtt{inj}_1\,\beta \Rightarrow \mathtt{Unit} \mid \mathtt{inj}_2\,\beta \Rightarrow \mathtt{Void}$ must in fact be unit. But this in turn tells us (informally) that $c$ can only be of the form $\mathtt{inj}_1\,\tau$.

So the informal reasoning of the previous paragraph tells us that after dispatching on the dynamic encoding of a type, we can informally infer information about its static encoding. In particular, if the dynamic encoding is a left injection, then the static encoding must similarly be a left injection; and similarly for right injections. In fact, as we shall see, the `vcase` constructs gives us a formal method for using this information.

Dynamic encodings of types are terms whose types are given by the operation of $R$ on their static encoding. So for example, the dynamic encodings of boxed 64-bit floats and of 32-bit ints are given as follows:

$$\begin{array}{lll} V_{BF} & :R(C_{BF}) & \stackrel{\mathrm{def}}{=} \;\; \mathtt{inj}_1 * \\ V_{I} & :R(C_{I}) & \stackrel{\mathrm{def}}{=} \;\; \mathtt{inj}_2 * \end{array}$$

### 4.2.3 Type analyzing terms

To re-cap, static encodings of types (such as $C_{BF}$) provide type level representations of the abstract syntax of the analyzable types. Dynamic encodings of types (such as $V_{BF}$) provide term level representations of the abstract syntax of the analyzable types, and are classified by types which depend on the static encoding of the type.

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

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

The type argument and the first term argument correspond to the static and dynamic encodings of the type of the contents of the array. The type of the initial value for the array is obtained by interpreting the encoded type using the interp function, and the return type is given by interpreting the encoded type using the $\texttt{Array}_{\mathrm{opt}}$ function.

The actual implementation of the array creation function simply case analyzes the dynamic encoding to determine the appropriate primitive array operation to use. The code itself is straightforward: the subtlety lies in understanding why the code is well-typed.

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

As expected, the type argument $\alpha$ has the kind of encoded types, $\mathrm{T}_{\mathrm{opt}}$. Similarly, the term argument $x_\alpha$ has the type of dynamic encodings of $\alpha$, and will be instantiated with the dynamic encoding of $\alpha$. 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($\alpha$). Similarly, the return type $\texttt{Array}_{\mathrm{opt}}(\alpha)$ gives the type of the returned array as a function of what $\alpha$ turns out to be.

The key to understanding how this all works out 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 $x_\alpha$, where the body of the arm does a virtual case analysis of $\alpha$. 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 $\alpha$, *including in the context*. This means that whereas outside the arm the variable $r$ has type

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

within the arm it has type

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

which is equivalent to simply `Void`. This satisfies the typing 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` |
|---|---|
| $\mathbf{y}$: $\texttt{case}\,\alpha\,\texttt{of}\quad \texttt{inj}_1\,\beta \Rightarrow \texttt{Boxed}(\beta)$ <br> $\qquad\qquad\mid \texttt{inj}_2\,\beta \Rightarrow \beta$ | $\mathbf{y}$: $\texttt{Boxed}(\beta_1)$ |
| $\mathbf{r}$: $\texttt{case}\,\alpha\,\texttt{of}\ \texttt{inj}_1\beta \Rightarrow \texttt{Unit} \mid \texttt{inj}_2\beta \Rightarrow \texttt{Void}$ | $\mathbf{r}$:`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.

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

# Chapter 5

# 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. The type analysis methodology for this is essentially the same as that described in the previous chapter, but extended to handle additional type analyzing operations.

The first section of this chapter gives a high-level overview of the translation, introducing the relations and stating some of the major typing properties. These translations are developed in detail in successive sections, along with proofs of their soundness.

## 5.1 Translation overview

The translation of **MIL** programs in to **LIL** programs is defined by several inductively defined relations between elements of the **MIL** syntactic classes and their correspondents in the **LIL**. These relations may be broadly grouped into four classes: those concerned with the static encoding of constructors, those concerned with the dynamic encoding of constructors, those concerned with proper **MIL** types, and those concerned with the expression translation itself.

The first group consists of the kind and constructor translations, and the corresponding translation on constructor typing contexts.

| Static encoding translations | | |
|---|---|---|
| **SEK** | $\lvert\kappa\rvert$ | $\bullet \vdash \lvert\kappa\rvert$ **ok** |
| **SE** context | $\lvert\Delta\rvert$ | $\vdash \lvert\Delta\rvert$ **ok** |
| **SE** | $\lvert c\rvert$ | $\lvert\Delta\rvert \vdash \lvert c\rvert : \lvert\kappa\rvert$ |

The kind translation replaces **MIL** kinds (closed by definition) with closed **LIL** kinds classifying the translation of **MIL** constructors of the original kind. The static encoding translation for contexts ($\lvert\Delta\rvert$) simply applies the kind translation across the range of a **MIL** typing context. This allows the statement of the desired typing property of the constructor translation: that if $\Delta \vdash c : \kappa$, then $\lvert\Delta\rvert \vdash \lvert c\rvert : \lvert\kappa\rvert$ (theorem 2). (There are of course a number of other auxiliary typing properties to be shown: these are covered in more detail in subsequent sections.)

The second group, primarily concerned with the dynamic encoding of constructors, consists of additional translations on constructors and typing contexts and a relation on constructors and kinds.

| Dynamic encoding translations | | |
|---|---|---|
| **DET** | $\lfloor c{:}\kappa \rfloor$ | $\lvert\Delta\rvert \vdash \lfloor c{:}\kappa \rfloor : \mathrm{T}_{32}$ |
| **DE** context | $\lfloor \Delta \rfloor$ | $\lvert\Delta\rvert \vdash \lfloor \Delta \rfloor$ **ok** |
| **DE** | $\lfloor c \rfloor$ | $\bullet; \lvert\Delta\rvert; \lfloor\Delta\rfloor \vdash \lfloor c \rfloor : \lfloor c{:}\kappa \rfloor$ **exp** |

The first translation, $\lfloor c{:}\kappa \rfloor$, is a relation between **MIL** constructor/kind pairs and **LIL** types. It defines the dynamic encoding type for a constructor $c$ of kind $\kappa$. The inhabitants of this type are the dynamic encodings of constructors, and are produced by a relation between **MIL** constructors and **LIL** expressions: $\lfloor c \rfloor$. The dynamic encoding translation for contexts maps **LIL** constructor contexts to **MIL** term contexts. The domain of the new context is constructed using an injection from type variables to term variables, while the range is constructed using the static encoding type translation. It should be the case that if $\Delta \vdash c : \kappa$, then $\Psi; \lvert\Delta\rvert; \lfloor\Delta\rfloor \vdash \lfloor c \rfloor : \lfloor c{:}\kappa \rfloor$ **exp** (theorem 3)

The third group is concerned with the translation of proper **MIL** types (as opposed to constructors).

| Type translations | | |
|---|---|---|
| Types | $\lvert\tau\rvert$ | $\lvert\Delta\rvert \vdash \lvert\tau\rvert : \mathrm{T}_{32}$ |
| Term context | $\lvert\Gamma\rvert$ | $\lvert\Delta\rvert \vdash \lvert\Gamma\rvert$ **ok** |

The type translation is a relation between proper **MIL** types and **LIL** constructors of kind $\mathrm{T}_{32}$, and the term context translation simply maps this translation across the range of **MIL** term contexts. It should be the case that if $\Delta \vdash \tau$ **ok**, then $\lvert\Delta\rvert \vdash \lvert\tau\rvert : \mathrm{T}_{32}$.

The final group of translations relates **MIL** terms of various syntactic classes to their corresponding **LIL** terms.

| Term translations | | |
|---|---|---|
| Small values | $\Delta; \Gamma \vdash sv : \tau \rightsquigarrow sv'$ | $\Psi; \lvert\Delta\rvert; \lfloor\Delta\rfloor, \lvert\Gamma\rvert \vdash sv' : \lvert\tau\rvert$ |
| Float values | $\Delta; \Gamma \vdash fv : \mathtt{Float} \rightsquigarrow fv'$ | $\Psi; \lvert\Delta\rvert; \lfloor\Delta\rfloor, \lvert\Gamma\rvert \vdash fv' : \mathtt{Float}$ |
| Operations | $\Delta; \Gamma \vdash opr : \tau \rightsquigarrow opr'$ | $\Psi; \lvert\Delta\rvert; \lfloor\Delta\rfloor, \lvert\Gamma\rvert \vdash opr' : \lvert\tau\rvert$ $\mathbf{opr}_{32}$ |
| 64 bit Operations | $\Delta; \Gamma \vdash fopr : \mathtt{Float} \rightsquigarrow fopr'$ | $\Psi; \lvert\Delta\rvert; \lfloor\Delta\rfloor, \lvert\Gamma\rvert \vdash fopr' : \mathtt{Float}$ $\mathbf{opr}_{64}$ |
| Expression | $\Delta; \Gamma \vdash e : \tau \rightsquigarrow e'$ | $\Psi; \lvert\Delta\rvert; \lfloor\Delta\rfloor, \lvert\Gamma\rvert \vdash e' : \lvert\tau\rvert$ **exp** |

It is convenient to phrase these as typed translations since the additional type information is sometimes required in order to construct the appropriate **LIL** syntax. The intended typing properties of these translations should be clear. For example, it should be the case that if $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \vdash e : \tau \rightsquigarrow e'$ then $\Psi; \lvert\Delta\rvert; \lfloor\Delta\rfloor, \lvert\Gamma\rvert \vdash e' : \lvert\tau\rvert$ **exp** (for appropriate heap contexts $\Psi$, theorem 8).

## 5.2 Static encodings of constructors

.

A static encoding is a **LIL** constructor which encodes information about the **MIL** constructor which it represents. The static encoding 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. This section develops these mechanisms, beginning with the translation of kinds.

### 5.2.1 The kind translation

The choice of what information to capture in the encoding depends entirely on the kind of type analysis optimization that is desired. In the **MIL**, there are two major uses of type analysis: an array flattening optimization and the *vararg* optimization. The array optimization is almost exactly as described in chapter 4, and requires the encoding strategy to distinguish between boxed floats and other types. The vararg optimization is described in chapter 2, and requires the ability to distinguish between records of various widths and other types so that the **vararg** and **onearg** operations can choose an appropriate calling convention. For the purposes of this translation then, it is sufficient for the encoding to capture three classes of types: records (with their widths), boxed floating point numbers, and all other types. In fact, the actual implementation sub-divides the last category into pointer types and non-pointer types in order to implement a further optimization on sums that is not discussed here, as it adds nothing substantial to the discussion.

This division of types into categories is apparent in the static encoding kind that we choose for the translation:

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

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

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

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

Recall that the **SEK** is the kind that classifies static encodings. What this definition tells us is that encoded constructors are either a list of types corresponding to the types of the fields of a record, a boxed floating point number, or some other unknown type. Note that since the only 64 bit type in the **MIL** is `Float`, it is not necessary to include any information in the second arm of the sum. If we wished to allow arbitrary 64 bit types to be flattened into arrays, we could replace the 1 in the second arm of the sum with a $\mathrm{T_{64}}$.

The previous definition tells us what the kind of encodings of proper **MIL** types looks like. Arbitrary **MIL** kinds are translated into **LIL** kinds simply by replacing all occurrences of $\mathrm{T_{32}}$ with $\mathrm{T_{mil}}$, and leaving the rest of the structure intact.

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

Following the methodology described in chapter 4, the first important object level type function we must define to assist with the encoding is an interpretation function which captures the meaning of the encoding in terms of the underlying types. This is done by defining an interpretation function interp of kind $\mathrm{T_{mil}} \to \mathrm{T_{32}}$.

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

For clarity, I use an ML style pattern matching notation based on the informal datatype definition given above. It should be completely clear how to translate this to formal syntax simply by replacing uses of **Record** $l$, for example, with $\mathtt{inj}_1\, l$.

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.

### 5.2.2 The constructor translation

The actual constructor translation $|c|$ translates **MIL** constructors of kind $\kappa$ to **LIL** constructors of kind $|\kappa|$. This means that constructors of kind $\mathrm{T}_{32}$ will be mapped to **LIL** constructors of kind $\mathrm{T}_{\mathrm{mil}}$. For clarity in the translation, I begin by defining some **LIL** functions that serve to construct static encodings of types. These serve as "constructors" for the ML style notation suggested by the datatype given above.

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

Note that for syntactic clarity I frequently leave off the kind decoration on sum injections where it is obvious from context. Frequently, throughout this dissertation I will use boldface for names of defined forms such as these to distinguish them from primitive syntax.

The actual static encoding translation proceeds for the most part compositionally over the structure of constructors. Constructors of higher kind are translated directly by recursively translating their sub-components. All other constructors are encoded into the $\mathrm{T}_{\mathrm{mil}}$ kind.

It is convenient to define object level functions for use in the translation and elsewhere that correspond exactly to the type analyzing primitives from the **MIL**: $\mathtt{Array}_c$ and $\mathtt{Vararg}_{c_1 \to c_2}$. This should again be familiar from the methodology developed in chapter 4.

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. Note that in the record case, it must explicitly reconstruct the record using the constituent field types.

$$\begin{array}{lll}
\textbf{Array} & :\mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32} \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{\mathrm{mil}}).\mathtt{case}\,\alpha & \\
& & \textbf{Record}\ \beta \Rightarrow \mathtt{Array}_{32}(\times(\beta)) \\
& & |\,\textbf{BFloat} \Rightarrow \mathtt{Array}_{64}(\mathtt{Float}) \\
& & |\,\textbf{Other}\ t \Rightarrow \mathtt{Array}_{32} t
\end{array}$$

`Vararg` types similarly become dispatches over the argument types to select the appropriate function type: either a flattened function type for small records; otherwise a standard record. Throughout this translation, we assume that only records with fewer than 3 fields get flattened into

$$
\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 \\
|\texttt{Int}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{Int}) \\
|\,\texttt{Sum}_i(c_i,\ldots,c_n)| & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\quad \mathbf{Other}(\bigvee[\texttt{Tag}(0),\ldots,\texttt{Tag}(i-1), \times[\texttt{Tag}(i), \text{interp}\,|c_i|],\ldots,\times[\texttt{Tag}(n),\text{interp}\,|c_n|]])} \\
|\,\texttt{Sum}_i^j(c_i,\ldots,c_n)| & \stackrel{\text{def}}{=} & \mathbf{Other}(\texttt{Tag}(j)) \quad j < i \\
|\,\texttt{Sum}_i^j(c_i,\ldots,c_j,\ldots,c_n)| & \stackrel{\text{def}}{=} & \mathbf{Other}(\times[\texttt{Tag}(j), \text{interp}\,|c_j|]) \quad i \le j \le 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}(\mathbf{Array}(|c|)) \\
|\texttt{Vararg}_{c_1 \to c_2}| & \stackrel{\text{def}}{=} & \mathbf{Other}(\mathbf{Vararg}(|c_1|)(\text{interp}\,|c_2|)) \\
|\mu(\alpha,\beta).(c_1,c_2)| & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{l}{\qquad\qquad \langle \mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_1 *)), \mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_2 *)\rangle} \\
\multicolumn{3}{l}{\qquad\qquad\quad \text{where} f = \lambda(\rho{:}1+1 \to \mathrm{T}_{32}).\lambda(\omega.1+1).} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \texttt{case}\,\omega} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\quad \texttt{inj}_1\,\_ \Rightarrow \text{interp}(|c_1|[\mathbf{Other}(\rho(\texttt{inj}_1 *))/\alpha, \mathbf{Other}(\rho(\texttt{inj}_2 *))/\beta])} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\quad |\,\texttt{inj}_2\,\_ \Rightarrow \text{interp}(|c_2|[\mathbf{Other}(\rho(\texttt{inj}_1 *))/\alpha, \mathbf{Other}(\rho(\texttt{inj}_2 *))/\beta])}
\end{array}
$$

**Figure 5.1:** The constructor translation (static encoding)

registers, but in general the number of fields is a machine dependent parameter of the translation.

$$
\begin{aligned}
\textbf{Vararg} \quad &: \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32} \to \mathrm{T}_{32} \stackrel{\mathrm{def}}{=} \lambda(\alpha{:}\mathrm{T}_{\mathrm{mil}}).\lambda(\beta{:}\mathrm{T}_{32}).\mathtt{case}\,\alpha \\
&\quad \textbf{Record}\ l \Rightarrow \\
&\quad\ (\mathtt{case}\, l \\
&\quad\quad\ [\,] \Rightarrow () \to \beta \\
&\quad\quad |[t] \Rightarrow (t) \to \beta \\
&\quad\quad |[t_1, t_2] \Rightarrow (t_1, t_2) \to \beta \\
&\quad\quad |\_ \Rightarrow (\times(l)) \to \beta) \\
&\quad |\,\textbf{BFloat} \Rightarrow (\mathtt{Boxed\,Float}) \to \beta) \\
&\quad |\,\textbf{Other}\ t \Rightarrow (t) \to \beta
\end{aligned}
$$

Note that the **Vararg** function as defined above is asymmetric in its two arguments: it expects the static encoding of the argument type, but for the return type expects the type itself. This reflects the fact that the result depends only on the argument type and not the result type. An alternative (and equally valid) definition for the `Vararg` primitive can be given that expects static encodings for both arguments (since after all, the argument passed to the **Vararg** function will almost always be the interpretation of a static encoding).

$$
\begin{aligned}
\textbf{Vararg} \quad &: \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32} \stackrel{\mathrm{def}}{=} \lambda(\alpha{:}\mathrm{T}_{\mathrm{mil}}).\lambda(\beta{:}\mathrm{T}_{\mathrm{mil}}).\mathtt{case}\,\alpha \\
&\quad \textbf{Record}\ l \Rightarrow \\
&\quad\ (\mathtt{case}\, l \\
&\quad\quad\ [\,] \Rightarrow () \to \mathrm{interp}(\beta) \\
&\quad\quad |[t] \Rightarrow (t) \to \mathrm{interp}(\beta) \\
&\quad\quad |[t_1, t_2] \Rightarrow (t_1, t_2) \to \mathrm{interp}(\beta) \\
&\quad\quad |\_ \Rightarrow (\times(l)) \to \mathrm{interp}(\beta)) \\
&\quad |\,\textbf{BFloat} \Rightarrow (\mathtt{Boxed\,Float}) \to \mathrm{interp}(\beta)) \\
&\quad |\,\textbf{Other}\ t \Rightarrow (t) \to \mathrm{interp}(\beta))
\end{aligned}
$$

From a semantic standpoint, the two definitions are equivalent. In practice however, using the previous definition is likely to provide noticeable performance benefits in type-checking since it provides a definition site for the interpretation of the result type. In the absence of this, a graph reduction implementation is required to avoid repeated reductions. While this is certainly possible (and may be desired in any case), careful design of the defined forms used in the translation can yield substantial performance benefits for very small cost in effort. While I will for the most part defer discussion of implementation issues until part 2 of this thesis, it is nonetheless worth pointing out briefly here that a careful theoretical design can greatly impact the ease and efficiency of the implementation.

Using these definitions, the constructor translation itself (figure 5.1) is almost completely straightforward. For the most part, the translation proceeds compositionally over the constructors in the obvious manner, applying the static encoding constructors at the leaves of the syntax tree. The case for sum types is more interesting, since it makes the representation of sums as tagged unions explicit. Note also that the known sum type from the **MIL** disappears. Known sums are no longer necessary because the tagging has been made explicit, and hence the de-structuring of sum values can be done via record selection, instead of via the `proj` primitive.

The most syntactically complex rule of the translation defines the compulation of **MIL** style recursive types into **LIL** style parametric recursive types. The essential idea behind this translation is to replace uses of multiple mutually recursive types with the fix point of a single parameterized constructor. The first parameter of the constructor (as usual) is the recursive variable. The second parameter serves as a selector parameter which indicates which field of the original mutually recursive type is desired. The translation therefore proceeds by translating the bodies of the recursive types, replacing uses of the multiple mutually recursive variables with partial applications of the new recursive variable to appropriate sum injections indicating the selection of the appropriate arm. The body of the function of which a fixpoint is to be taken uses its parameter to select the appropriate arm of the recursive type to return. The final tuple of constructors is created by instantiating the fixpoint once for each of the different arms of the original constructor.

### 5.2.3 Translation of typing contexts

The kind translation extends in a natural way to define a translation on **MIL** constructor level typing contexts.

$$\begin{array}{lcl} |\bullet| & \stackrel{\text{def}}{=} & \bullet \\ |\Delta, \alpha{:}\kappa| & \stackrel{\text{def}}{=} & |\Delta|, \alpha{:}|\kappa| \end{array}$$

### 5.2.4 Proofs of soundness for the constructor and kind translation

**Lemma 4 (Well-formedness of $\mathrm{T_{mil}}$)**
*For all well formed $\Delta$ such that the bound variables $\mathrm{T_{mil}}$ are not in $\Delta$,*

$$\Delta \vdash \mathrm{T_{mil}}\ \mathbf{ok}$$

**Proof:** By construction. Note that for any given $\Delta$, the definitions may be alpha-varied such that the variable condition is satisfied.

■

**Lemma 5 (Soundness of kind translation (1))**
$\bullet \vdash |\kappa|\ \mathbf{ok}$

**Proof:** By induction over the structure of $\kappa$, and lemma 4.

■

**Corollary 1**
*If* $\vdash \Delta\ \mathbf{ok}$ *then* $\Delta \vdash |\kappa|\ \mathbf{ok}$.

**Proof:** By lemma 5 and weakening.

■

**Lemma 6**
*If* $\alpha \notin \Delta'$ *then* $\alpha \notin |\Delta|$.

**Proof:** Note that $\Delta$ and $|\Delta|$ have the same domain by construction.

■

**Lemma 7 (Soundness of the context translation)**
*If* $\vdash \Delta\ \mathbf{ok}$ *then* $\vdash |\Delta|\ \mathbf{ok}$.

**Proof:** By induction over the structure of $\Delta$.

1. If $\Delta = \bullet$ then $|\Delta| = \bullet$ and $\vdash \bullet$ **ok**

2. If $\Delta = \Delta', \alpha{:}\kappa$ then:

   By assumption:
   $\vdash \Delta'$ **ok**, $\alpha \notin \Delta'$, and $\vdash \kappa$ **ok**

   By induction:
   $\vdash |\Delta'|$ **ok**

   By corollary 1:
   $|\Delta'| \vdash |\kappa|$ **ok**

   By lemma 6:
   $\alpha \notin |\Delta'|$

   By construction:
   $\vdash |\Delta'|, \alpha{:}|\kappa|$ **ok**

   By definition:
   $\vdash |\Delta|$ **ok**

■

The previous result is used to prove a slightly stronger result about the soundness of the kind translation.

**Theorem 1 (Soundness of kind translation (2))**
*If* $\vdash \Delta$ **ok** *then* $|\Delta| \vdash |\kappa|$ **ok**.

**Proof:** By lemma 7, $\vdash |\Delta|$ **ok**, so by corollary 1, $|\Delta| \vdash |\kappa|$ **ok**.

■

To assist in the proof of soundness of the constructor translation, I formalize the typing properties of the definitions in the following lemma:

**Lemma 8 (Well-formedness of definitions)**
*For all well formed* $\Delta$ *such that the bound variables of the respective defined forms are not in* $\Delta$,

$$\begin{array}{l}
\Delta \vdash \text{interp} : \mathrm{T}_{\text{mil}} \to \mathrm{T}_{32} \\
\Delta \vdash \textbf{Other} : \mathrm{T}_{32} \to \mathrm{T}_{\text{mil}} \\
\Delta \vdash \textbf{BFloat} : \mathrm{T}_{\text{mil}} \\
\Delta \vdash \textbf{Record} : \mathrm{T}_{32}\texttt{list} \to \mathrm{T}_{\text{mil}} \\
\Delta \vdash \textbf{Array} : \mathrm{T}_{\text{mil}} \to \mathrm{T}_{32} \\
\Delta \vdash \textbf{Vararg} : \mathrm{T}_{\text{mil}} \to \mathrm{T}_{32} \to \mathrm{T}_{32}
\end{array}$$

**Proof:** By construction. Note that for any given $\Delta$, the definitions may be alpha-varied such that the variable condition is satisfied.

■

Using this lemma, the soundness of the constructor translation follows straightforwardly.

**Theorem 2 (Soundness of the constructor translation)**
*If* $\Delta \vdash c : \kappa$ *then* $|\Delta| \vdash |c| : |\kappa|$.

**Proof:** By induction on the structure of typing derivations.

1. Suppose $\Delta, \alpha{:}\kappa \vdash \alpha : \kappa$. Then by induction $\vdash |\Delta, \alpha{:}\kappa|$ **ok**. By definition $|\Delta, \alpha{:}\kappa| = |\Delta|, \alpha{:}\kappa$, and $|\alpha| = \alpha$, and so by construction $|\Delta, \alpha{:}\kappa| \vdash \alpha : \kappa$.

2. Suppose $\Delta \vdash \lambda(\alpha{::}\kappa_1).c : \kappa_1 \to \kappa_2$. By theorem 1, $\Delta \vdash |\kappa_1|$ **ok**, and by induction, $|\Delta, \alpha{:}\kappa_1| \vdash |c| : |\kappa_2|$. But $|\Delta, \alpha{:}\kappa_1| = |\Delta|, \alpha{:}|\kappa_1|$, and by lemma 6, $\alpha \notin |\Delta|$. So by construction, $|\Delta| \vdash |\lambda(\alpha{::}\kappa_1).c| : |\kappa_1 \to \kappa_2|$.

3. Suppose $\Delta \vdash c_1 c_2 : \kappa_2$. By induction, $|\Delta| \vdash |c_1| : |\kappa_1 \to \kappa_2|$ and $|\Delta| \vdash |c_2| : |\kappa_1|$. Therefore by construction $|\Delta| \vdash |c_1 c_2| : |\kappa_2|$.

4. Suppose $\Delta \vdash \pi_1\, c : \kappa_1$. By induction, $|\Delta| \vdash |c| : |\kappa_1 \times \kappa_2|$. So by construction, $|\Delta| \vdash |\,\pi_1\, c| : |\kappa_1|$.

5. Suppose $\Delta \vdash \pi_2\, c : \kappa_2$. By a similar argument to the previous case, $|\Delta| \vdash |\,\pi_2\, c| : |\kappa_2|$.

6. Suppose $\Delta \vdash \langle c_1, c_2 \rangle : \kappa_1 \times \kappa_2$. By induction, $|\Delta| \vdash |c_1| : |\kappa_1|$ and $|\Delta| \vdash |c_2| : |\kappa_2|$. So by construction, $|\Delta| \vdash |\langle c_1, c_2 \rangle| : |\kappa_1 \times \kappa_2|$

7. Suppose $\Delta \vdash \texttt{Int} : \mathrm{T}_{32}$. Note that $|\mathrm{T}_{32}| = \mathrm{T}_{\mathrm{mil}}$, so it suffices to show that $|\Delta| \vdash |\texttt{Int}| : \mathrm{T}_{\mathrm{mil}}$. By lemma 8, $|\Delta| \vdash \mathbf{Other} : \mathrm{T}_{32} \to \mathrm{T}_{\mathrm{mil}}$, and by the `Int` axiom, $|\Delta| \vdash \texttt{Int} : \mathrm{T}_{32}$. So by the application rule, $|\Delta| \vdash \mathbf{Other}(\texttt{Int}) : \mathrm{T}_{\mathrm{mil}}$.

8. The other primitive type constructors follow similarly as with `Int`.

9. Suppose $\Delta \vdash \mu(\alpha, \beta).(c_1, c_2) : \mathrm{T}_{32} \times \mathrm{T}_{32}$. It suffices to show that

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

But note that it suffices to show that $|\Delta| \vdash f : (1+1 \to \mathrm{T}_{32}) \to (1+1) \to \mathrm{T}_{32}$, since the desired derivation can then be produced by applying the `rec` rule to show the well-formedness of the new recursive types; the application rule and lemma 8 to show the well-formedness of the applications of **Other**, and the pairing rule to show the well-formedness of the pair.

By induction, $|\Delta, \alpha{:}\mathrm{T}_{32}, \beta{:}\mathrm{T}_{32}| \vdash |c_1| : \mathrm{T}_{\mathrm{mil}}$ and by weakening, the freshness assumption, and the definition of the context translation $|\Delta|, \alpha{:}\mathrm{T}_{\mathrm{mil}}, \beta{:}\mathrm{T}_{\mathrm{mil}}, \rho{:}(1+1 \to \mathrm{T}_{32}) \vdash |c_1| : \mathrm{T}_{\mathrm{mil}}$. By the substitution lemma for the **LIL** (lemma 1):

$$|\Delta|, \rho{:}(1+1 \to \mathrm{T}_{32}) \vdash |c_1|[\mathbf{Other}(\rho(\texttt{inj}_1 *))/\alpha, \mathbf{Other}(\rho(\texttt{inj}_2 *))/\beta] : \mathrm{T}_{\mathrm{mil}}$$

By the application typing rule therefore:

$$|\Delta|, \rho{:}(1+1 \to \mathrm{T}_{32}) \vdash \mathrm{interp}(|c_1|[\mathbf{Other}(\rho(\texttt{inj}_1 *))/\alpha, \mathbf{Other}(\rho(\texttt{inj}_2 *))/\beta]) : \mathrm{T}_{32}$$

A similar argument holds for $c_2$. Therefore, by applying the typing rule for case and for repeated lambda abstraction (again using freshness and weakening for $\omega$), we construct a well-formedness derivation for $f$.

■

### 5.2.5 Additional properties of the constructor translation

#### Commutation with substitution

It is an additional property of the constructor translation that it commutes with substitution: i.e. that it is compositional. This property is necessary for the proofs of a number of subsequent theorems, and so I give a proof of it here.

**Lemma 9 (The constructor translation commutes with substitution)**
$|c|[|c'|/\alpha] = |c[c'/\alpha]|$.

**Proof:** By induction on $c$.

1. If $c$ is a constant, then $\alpha \notin fvc, fv|c|$, so $|c|[|c'|/\alpha] = |c| = |c[c'/\alpha]|$.
2. If $c$ is a variable $\alpha'$:
   (a) If $\alpha \neq \alpha'$ then as in the previous case.
   (b) If $\alpha = \alpha'$ then $|\alpha|[|c'|/\alpha] = \alpha[|c'|/\alpha] = |c'| = |\alpha[c'/\alpha]|$.
3. If $c = \lambda(\beta{:}\kappa).c_2$, then by definition

   $$|\lambda(\beta{:}\kappa).c_2|[c'/\alpha] = (\lambda(\beta{:}|\kappa|).|c_2|)[c'/\alpha] = \lambda(\beta{:}|\kappa|).(|c_2|[c'/\alpha])$$

   By induction and the definition of substitution:

   $$\lambda(\beta{:}|\kappa|).(|c_2|[c'/\alpha]) = \lambda(\beta{:}|\kappa|).(|c_2[c'/\alpha]|) = |\lambda(\beta{:}\kappa).(c_2[c'/\alpha])|$$

   Note, we rely on alpha-variance to ensure non-capture.
4. If $c = c_1 c_2, \pi_i c, \langle c_1, c_2\rangle$, the proof proceeds similarly.

■

#### The SE translation respects equivalence

Another important property of the **SE** translation is that equivalent **MIL** constructors translate to equivalent **LIL** constructors. In addition to being important for subsequent proofs, this lemma is an important "sanity-check" on the translation.

**Lemma 10 (The constructor translation respects equivalence)**
*If* $\Delta \vdash c_1 \equiv c_2 : \kappa$, *then* $|\Delta| \vdash |c_1| \equiv |c_2| : |\kappa|$.

**Proof:** (By induction on equivalence derivations) All of the structural and type constructor equivalence rules are unchanged from the **MIL** to the **LIL**, and the proof follows straightforwardly by induction. I give the reflexivity rule, the pair rule and the beta rule as examples. All of the primitive types follow directly by reflexivity or by the structural application rule.

1. Suppose $\Delta \vdash c \equiv c : \kappa$ by reflexivity.

   By assumption:
   $\Delta \vdash c : \kappa$

By theorem 2:
  $|\Delta| \vdash |c| : |\kappa|$

By reflexivity:
  $|\Delta| \vdash |c| \equiv |c| : |\kappa|$

2. Suppose $\Delta \vdash (\lambda(\alpha{:}\kappa_1).c_1)(c_2) \equiv c_1[c_2/\alpha] : \kappa_2$.

   By assumption:
   $\vdash \kappa_1$ **ok**
   $\Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa_2$
   $\Delta \vdash c_2 : \kappa_1$

   By theorems 1, and 2:
   $|\Delta| \vdash |\kappa_1|$ **ok**
   $|\Delta, \alpha{:}\kappa_1| \vdash |c_1| : |\kappa_2|$
   $|\Delta| \vdash |c_2| : |\kappa_1|$

   By the $\lambda$ beta rule and the definition of the translations:
   $|\Delta| \vdash |(\lambda(\alpha{:}\kappa_1).c_1)(c_2)| \equiv |c_1|[|c_2|/\alpha] : |\kappa_2|$

   Finally, by lemma 9:
   $|\Delta| \vdash |(\lambda(\alpha{:}\kappa_1).c_1)(c_2)| \equiv |c_1[c_2/\alpha]| : |\kappa_2|$

■

## 5.3 Dynamic encoding of constructors

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

A constructor's dynamic encoding 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. Dynamic encodings are used to implement type dispatch at runtime. This section defines the dynamic encoding translation.

### 5.3.1 Dynamic encoding types

The dynamic encoding translation of a constructor $c$ is notated as $\rfloor c \lfloor$. (The notation is intended to be suggestive of the fact that the translation moves "down" a level, from constructors to terms.) The type of the dynamic encoding 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 encoding of a constructor also depends on its static encoding: this is what captures the connection between static and dynamic encodings. For a **MIL** kind $\kappa$ classifying a constructor $c$, I notate the dynamic encoding type for the constructor as $\rfloor c{:}\kappa \lfloor$.

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

The **DET** translation is defined in terms of a type level function $R$ of type $\mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32}$. This function is defined shortly, but for the sake of understanding the translation it is easier to view this abstractly as the type of representations of constructors of kind $\mathrm{T}_{32}$.

For constructors of pair kind, the **DET** is straightforward: a pair type with fields constructed by projecting out the two halves of the pair and applying the **DET** translation to each of them at the sub-component kind.

Constructor functions are slightly more interesting. The essential idea is that constructor functions (functions from types to types) will become term functions (functions from terms to terms). Accordingly, the **DET** of a constructor of arrow kind is a term level function type. However, note that this requires us to construct the **DET** of the argument type and the result type of the function, which in turn requires us to have the static encoding of the argument to the type function (which of course is not yet available). The solution to this dilemma is to observe that the **DE** function in question must be *polymorphic* over all possible static encodings: the **DET** of the argument and result can then be given in terms of the eventual encoded type passed to the function. This point is essential to the methodological goal of maintaining the connection between static encodings and dynamic encodings.

So far, I have avoided discussing the particulars of the actual encodings of types (that is, of constructors of kind $\mathrm{T}_{32}$). Here what is required is something of the nature of the representation types of $\lambda_r$ [CWM98]: the type of representation of a specific type. Unlike in $\lambda_r$, these types are not primitive here. Instead, they are programmed directly in the **LIL** in the following fashion.

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

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 $\texttt{case}\,\alpha\,(\mathbf{BFloat}\ \_ \Rightarrow \texttt{Unit} \mid \_ \Rightarrow \texttt{Void})$. Since the `Void` type is uninhabited, having a value of this type means that $\alpha$ 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 $\alpha$.

### 5.3.2 Notational issues

Unfortunately, the named-form syntactic restrictions imposed on **LIL** terms add a level of inessential complexity to the dynamic encoding translations. Constructors in the **MIL** are not in named-form: in fact, in the absence of singleton kinds or some other definitional mechanism there is in general no equivalent named-form for an arbitrary constructor. This means that translating **MIL** constructors into **LIL** terms requires the dynamic encoding essential to the translation to occur simultaneously

| Pairs of expressions $\langle e_1, e_2\rangle$ | |
|---|---|
| $\langle sv_1, sv_2\rangle$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = \langle sv_1, sv_2\rangle \,\texttt{in}\, x$ |
| $\langle sv_1, \texttt{let}\, x = i \,\texttt{in}\, e_2\rangle$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = i \,\texttt{in}\, \langle sv_1, e_2\rangle \quad (x \notin fv(sv_1))$ |
| $\langle \texttt{let}\, x = i \,\texttt{in}\, e_1, e_2\rangle$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = i \,\texttt{in}\, \langle e_1, e_2\rangle \quad (x \notin fv(e_2))$ |
| Projection from expressions $\texttt{select}^i\, e$ | |
| $\texttt{select}^i\, sv$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = \texttt{select}^i\, sv \,\texttt{in}\, x$ |
| $\texttt{select}^i(\texttt{let}\, x = i \,\texttt{in}\, e)$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = i \,\texttt{in}\, (\texttt{select}^i\, e)$ |
| Sum injection of expressions $\texttt{inj}_i^c\, e$ | |
| $\texttt{inj}_i^c\, sv$ | $\stackrel{\text{def}}{=} \texttt{let}\, y = \langle \texttt{tag}_i, sv\rangle \,\texttt{in let}\, x = \texttt{inj}_i^c\, y \,\texttt{in}\, x$ |
| $\texttt{inj}_i^c(\texttt{let}\, x = i \,\texttt{in}\, e)$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = i \,\texttt{in}\, (\texttt{inj}_i^c\, e)$ |
| Application of expressions $e_1[c]e_2$ | |
| $sv_1[c]sv_2$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = sv_1[c]sv_2 \,\texttt{in}\, x$ |
| $sv_1[c](\texttt{let}\, x = i \,\texttt{in}\, e_2)$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = i \,\texttt{in}\, (sv_1[c]e_2) \quad (x \notin fv(sv_1))$ |
| $(\texttt{let}\, x = i \,\texttt{in}\, e_1)[c]e_2$ | $\stackrel{\text{def}}{=} \texttt{let}\, x = i \,\texttt{in}\, (e_1[c]e_2) \quad (x \notin fv(e_2))$ |

**Figure 5.2:** Derived non-named-form expressions

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash e_1 : \tau_1 \ \mathbf{exp} \\ \Psi;\Delta;\Gamma \vdash e_2 : \tau_2 \ \mathbf{exp}\end{array}}{\Psi;\Delta;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2 \ \mathbf{dexp}} \qquad \frac{\Psi;\Delta;\Gamma \vdash e : \tau_0 \times \tau_1 \ \mathbf{exp} \quad (i \in 0, 1)}{\Psi;\Delta;\Gamma \vdash \texttt{select}^i\, e : \tau_i \ \mathbf{dexp}}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv +[\tau_1, \ldots, \times[\texttt{Tag}(i), \tau_i], \ldots, \tau_n] : \mathrm{T}_{32} \\ \Psi;\Delta;\Gamma \vdash e : \tau_i \ \mathbf{exp} \quad (i \in 1 \ldots n)\end{array}}{\Psi;\Delta;\Gamma \vdash \texttt{inj}_i^c\, e : c \ \mathbf{dexp}} \qquad \frac{\begin{array}{c}\Delta \vdash c : \kappa \\ \Psi;\Delta;\Gamma \vdash e_1 : \forall(\alpha{:}\kappa).\tau_1 \to \tau_2 \ \mathbf{exp} \\ \Psi;\Delta;\Gamma \vdash e_2 : \tau_1[c/\alpha] \ \mathbf{exp}\end{array}}{\Psi;\Delta;\Gamma \vdash e_1[c]e_2 : \tau_2[c/\alpha] \ \mathbf{dexp}}$$

**Figure 5.3:** Typing rules for derived (non-named-form) expressions

$$
\begin{array}{lcl}
\rfloor\alpha\lfloor & \stackrel{\text{def}}{=} & x_\alpha \\
\rfloor\lambda(\alpha{:}\kappa).c\lfloor & \stackrel{\text{def}}{=} & \lambda[\alpha{:}|\kappa|](x_\alpha{:}\ \rfloor\alpha{:}\kappa\lfloor).\ \rfloor c\lfloor \\
\rfloor c_1 c_2\lfloor & \stackrel{\text{def}}{=} & \rfloor c_1\lfloor\ [|c_2|](\rfloor c_2\lfloor) \\
\rfloor\pi_i\, c\lfloor & \stackrel{\text{def}}{=} & \texttt{select}_i\ \rfloor c\lfloor \\
\rfloor\langle c1, c2\rangle\lfloor & \stackrel{\text{def}}{=} & \langle\rfloor c_1\lfloor, \rfloor c_2\lfloor\rangle \\
\rfloor\texttt{Unit}\lfloor & \stackrel{\text{def}}{=} & \texttt{inj}_1^{R(|\texttt{Unit}|)}(\texttt{inj}_1\langle\rangle) \\
\rfloor\times[c]\lfloor & \stackrel{\text{def}}{=} & \texttt{inj}_1^{R(|\times[c]|)}(\texttt{inj}_2(\texttt{inj}_1\langle\rangle)) \\
\rfloor c_1\times c_2\lfloor & \stackrel{\text{def}}{=} & \texttt{inj}_1^{R(|c_1\times c_2|)}(\texttt{inj}_2(\texttt{inj}_2(\texttt{inj}_1\langle\rangle))) \\
\rfloor\times[c_1, c_2, \ldots, c_n]\lfloor & \stackrel{\text{def}}{=} & \texttt{inj}_1^{R(|\times[c_1,c_2,\ldots,c_n]|)}(\texttt{inj}_2(\texttt{inj}_2(\texttt{inj}_2\langle\rangle))) \\
\rfloor\texttt{Boxf}\lfloor & \stackrel{\text{def}}{=} & \texttt{inj}_2^{R(|\texttt{Boxf}|)}\langle\rangle \\
\rfloor c\ \texttt{as}\ \mu(\alpha,\beta).(c_1,c_2)\lfloor & \stackrel{\text{def}}{=} & \langle\texttt{inj}_3^{R(|\,\pi_1\, c|)}\langle\rangle, \texttt{inj}_3^{R(|\,\pi_2\, c|)}\langle\rangle\rangle \\
\rfloor c\ \textit{otherwise}\lfloor & \stackrel{\text{def}}{=} & \texttt{inj}_3^{R(|c|)}\langle\rangle
\end{array}
$$

**Figure 5.4:** The dynamic encoding translation

with a mostly un-interesting "lifting" process which pulls bindings out of expressions and returns them to named form.

In order to isolate the core issues of the translation from these syntactic issues, it is useful to define derived syntactic forms (figure 5.2) that allow the translation to make use of apparently arbitrary expressions without concerning itself with named form. Derived typing rules (figure 5.3) for the extended syntax with an accompanying proof of soundness (section 5.3.5) permit the free use of the extended syntax in the translation.

Only a small number of derived expression forms are needed for the purposes of the dynamic encoding translation: additional definitions are possible. All variables are assumed to be fresh in the translation: this guarantees that variables will not conflict as bindings are "lifted". This could also be dealt with by explicitly alpha-varying terms as necessary.

### 5.3.3 Dynamic encodings

The term-level encoding translation of constructors is given in figure 5.4 and is for the most part fairly intuitive. Note that the translation takes advantage of the defined forms from section 5.3.2 for presentational clarity: without this, it is necessary for the translation to handle explicitly the lifting of bindings. In point of fact, this sort of issue arises in many parts of the implementation of the translation from **MIL** to **LIL** and is handled using a monadic structure similar to the defined syntax used here.

For the purposes of the translation I assume an unspecified anti-symmetric injection from type variables into a set of term variables disjoint from those produced elsewhere in the translation. I refer to the term variables produced by this injection informally as being "indexed" by the type variable. Note that the translation does not rely on the ability to recover the original type variable from the indexed term variable.

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 $\mathrm{T}_{32}$, it is necessary to construct the type representations by injecting the appropriate witnesses into the sum type described above.

### 5.3.4 Dynamic encoding of type contexts

Since the **DE** translation is defined on open terms, the question naturally arises of what the appropriate notion of typing context for dynamic encodings should be. Just as the **DE** translation maps constructors to types, the corresponding context translation maps constructor contexts to term contexts using the **DET** translation defined above. Again, I use the down arrow syntax ($\downharpoonright\cdot\downharpoonleft$) to suggest the movement "down" a level in the syntactic hierarchy.

$$\begin{array}{lcl} \downharpoonright\bullet\downharpoonleft & \stackrel{\text{def}}{=} & \bullet \\ \downharpoonright\Delta, \alpha{:}\kappa\downharpoonleft & \stackrel{\text{def}}{=} & \downharpoonright\Delta\downharpoonleft, x_\alpha{:}\ \downharpoonright\alpha{:}\kappa\downharpoonleft \end{array}$$

Note that the types produced by the **DET** translation will contain references to the type variables from the original context: that is, the context produced by the translation will have free type variables. These variables are described by the context produced by the static encoding of the original context: that is, $|\Delta| \vdash \downharpoonright\Delta\downharpoonleft$ **ok**.

### 5.3.5 Proofs for the dynamic encoding translations

#### Proof of soundness of the DET translation

I begin by showing the well-formedness of the definition of representation types.

**Lemma 11 (Well-formedness of $R$)**
*For all well formed $\Delta$ such that the bound variables $R$ are not in $\Delta$, $\Delta \vdash R : \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32}$.*

**Proof:** By construction. Note that for any given $\Delta$, the definition may be alpha-varied such that the variable condition is satisfied.

∎

Using this and the soundness theorems for the **SE** and **SEK** translations, I show the soundness of the **DET** translation.

**Lemma 12 (Soundness of the dynamic encoding type translation)**
*If $\Delta \vdash c : \kappa$ then $|\Delta| \vdash \downharpoonright c{:}\kappa\downharpoonleft\ : \mathrm{T}_{32}$.*

**Proof:** By induction on kinds $\kappa$.

1. Suppose $\kappa = \mathrm{T}_{32}$. Then by definition, $\downharpoonright c{:}\kappa\downharpoonleft = R(|c|)$. By theorem 2, $|\Delta| \vdash |c| : \mathrm{T}_{\mathrm{mil}}$, and so by lemma 11 and the application typing rule, $|\Delta| \vdash R(|c|) : \mathrm{T}_{32}$.

2. Suppose $\kappa = \kappa_1 \to \kappa_2$. Then by definition, $\downharpoonright c{:}\kappa\downharpoonleft = \forall[\alpha{:}|\kappa_1|](\downharpoonright\alpha{:}\kappa_1\downharpoonleft) \to \downharpoonright c\alpha{:}\kappa_2\downharpoonleft$. It suffices to show that this is well-formed via the formation rule for universals. By theorem 1, $|\Delta| \vdash |\kappa_1|$ **ok**. By the variable rule, $\Delta, \alpha{:}\kappa_1 \vdash \alpha : \kappa_1$, and so by induction, $|\Delta, \alpha{:}\kappa_1| \vdash \downharpoonright\alpha{:}\kappa_1\downharpoonleft\ : \mathrm{T}_{32}$. Again using the variable rule along with the assumption, we obtain that $\Delta, \alpha{:}\kappa_1 \vdash c\alpha : \kappa_2$ by the application rule, and so by induction, $|\Delta, \alpha{:}\kappa_1| \vdash \downharpoonright c\alpha{:}\kappa_2\downharpoonleft\ : \mathrm{T}_{32}$.

We therefore may invoke the formation rule for universals to produce a derivation of the desired result.

3. Suppose $\kappa = \kappa_1 \rightarrow \kappa_2$. Then by definition, $\downharpoonleft c{:}\kappa \downharpoonright = \downharpoonleft \pi_1\, c{:}\kappa_1 \downharpoonright \times \downharpoonleft \pi_2\, c{:}\kappa_2 \downharpoonright$. It suffices to show that this is well-formed via the formation rule for pair types by showing that the two component types are well-formed. This follows immediately by induction.

■

As with the **SE** translation, the fact that the **DET** translation commutes with substitution is useful for subsequent proofs.

**Lemma 13 (The DET translation commutes with substitution)**
$\downharpoonleft c{:}\kappa \downharpoonright [|c'|/\alpha] = \downharpoonleft c[c'/\alpha]{:}\kappa \downharpoonright$.

**Proof:** By induction on kinds $\kappa$.

1. If $\kappa = \mathrm{T}_{32}$, then $\downharpoonleft c{:}\kappa \downharpoonright = R(|c|)$. Since $R$ is closed, $R(|c|)[|c'|/\alpha] = R(|c|[|c'|/\alpha])$. By lemma 9, $|c|[|c'|/\alpha] = |c[c'/\alpha]|$. Finally, by definition, $R(|c[c'/\alpha]|) = \downharpoonleft c[c'/\alpha]{:}\mathrm{T}_{32} \downharpoonright$.

2. If $\kappa = \kappa_1 \rightarrow \kappa_2$ then by definition

$$\begin{aligned} \downharpoonleft c{:}\kappa_1 \rightarrow \kappa_2 \downharpoonright [|c'|/\alpha] &\stackrel{\text{def}}{=} (\forall[\beta{:}|\kappa_1|](\downharpoonleft \beta{:}\kappa_1 \downharpoonright) \rightarrow (\downharpoonleft c\beta{:}\kappa_2 \downharpoonright))[|c'|/\alpha] \\ &\stackrel{\text{def}}{=} \forall[\beta{:}|\kappa_1|](\downharpoonleft \beta{:}\kappa_1 \downharpoonright) \rightarrow (\downharpoonleft c\beta{:}\kappa_2 \downharpoonright [|c'|/\alpha]) \end{aligned}$$

Note that $\alpha \neq \beta$ by the assumption of freshness in the translation, and hence

$$\downharpoonleft \beta{:}\kappa_1 \downharpoonright [|c'|/\alpha] = \downharpoonleft \beta{:}\kappa_1 \downharpoonright$$

By induction, $\downharpoonleft c\beta{:}\kappa_2 \downharpoonright [|c'|/\alpha] = \downharpoonleft (c[c'/\alpha])\beta{:}\kappa_2 \downharpoonright$, again using the fact that $\alpha \neq \beta$. Finally, we observe that

$$\forall[\beta{:}|\kappa_1|](\downharpoonleft \beta{:}\kappa_1 \downharpoonright) \rightarrow (\downharpoonleft (c[c'/\alpha])\beta{:}\kappa_2 \downharpoonright) = \downharpoonleft c[c'/\alpha]{:}\kappa_1 \rightarrow \kappa_2 \downharpoonright$$

3. If $\kappa = \kappa_1 \times \kappa_2$ then by definition,

$$\begin{aligned} \downharpoonleft c{:}\kappa_1 \times \kappa_2 \downharpoonright [|c'|/\alpha] &\stackrel{\text{def}}{=} (\downharpoonleft \pi_1 c{:}\kappa_1 \downharpoonright \times \downharpoonleft \pi_2 c{:}\kappa_2 \downharpoonright)[|c'|/\alpha] \\ &\stackrel{\text{def}}{=} (\downharpoonleft \pi_1 c{:}\kappa_1 \downharpoonright [|c'|/\alpha]) \times (\downharpoonleft \pi_2 c{:}\kappa_2 \downharpoonright [|c'|/\alpha]) \end{aligned}$$

By induction, $(\downharpoonleft \pi_i c{:}\kappa_i \downharpoonright [|c'|/\alpha]) = \downharpoonleft \pi_i c[c'/\alpha]{:}\kappa_i \downharpoonright$, so

$$\begin{aligned} (\downharpoonleft \pi_1 c{:}\kappa_1 \downharpoonright [|c'|/\alpha]) \times (\downharpoonleft \pi_2 c{:}\kappa_2 \downharpoonright [|c'|/\alpha]) &\stackrel{\text{def}}{=} \downharpoonleft \pi_1 c[c'/\alpha]{:}\kappa_1 \downharpoonright \times \downharpoonleft \pi_2 c[c'/\alpha]{:}\kappa_2 \downharpoonright \\ &\stackrel{\text{def}}{=} \downharpoonleft c[c'/\alpha]{:}\kappa_1 \times \kappa_2 \downharpoonright \end{aligned}$$

■

**Proof of soundness of the DE typing context translation**

**Lemma 14**
*If* $\alpha \notin \Delta$ *then* $x_\alpha \notin \lfloor\Delta\rfloor$.

**Proof:** By assumption, the injection from type variables to term variables is anti-symmetric with respect to variable equality. Therefore, by anti-symmetry, if $\beta \neq \alpha$ then $x_\beta \neq x_\alpha$. Since the set of variables in the domain of $\Delta$ does not contain $\alpha$, the image of the set of under the injection does not contain $x_\alpha$.

■

**Lemma 15 (Soundness of the context translation)**
*If* $\vdash \Delta$ **ok** *then* $|\Delta| \vdash \lfloor\Delta\rfloor$ **ok**.

**Proof:** By induction over the structure of $\Delta$.

1. If $\Delta = \bullet$ then $\lfloor\Delta\rfloor = \bullet$ and $\bullet \vdash \bullet$ **ok**
2. If $\Delta = \Delta', \alpha{:}\kappa$ then by assumption, $\vdash \Delta'$ **ok**, $\alpha \notin \Delta'$, and $\vdash \kappa$ **ok**. By the variable rule, $\Delta', \alpha{:}\kappa \vdash \alpha : \kappa$, and so by lemma 12, $|\Delta|, \alpha{:}|\kappa| \vdash \lfloor\alpha{:}\kappa\rfloor : \mathrm{T}_{32}$. By induction, $|\Delta'| \vdash \lfloor\Delta'\rfloor$ **ok**, and by weakening and lemma 6 $|\Delta'|, \alpha{:}|\kappa| \vdash \lfloor\Delta'\rfloor$ **ok**.

   Finally by the formation rule for contexts $|\Delta'|, \alpha{:}|\kappa| \vdash \lfloor\Delta'\rfloor, x_\alpha{:}\, \lfloor\alpha{:}\kappa\rfloor$ **ok**. Note that the side condition follows from lemma 14.

■

**Proof of soundness of the DE translation**

**Lemma 16 (Inversion of derived expression derivations)**
*For any derivation $D$ of $\Psi;\Delta;\Gamma \vdash e : \tau$* **dexp** *the last rule of $D$ is uniquely determined by $e$.*

**Proof:** By inspection. Note that for any choice of $e$, exactly one rule applies. ■

Since the **DE** translation relies on the extended syntax defined in section 5.3.2, is is first necessary to show that well-typedness as a derived form corresponds with well-typedness of expressions.

**Lemma 17 (Derived expression rules)**
*If* $\Psi;\Delta;\Gamma \vdash e : \tau$ **dexp** *then* $\Psi;\Delta;\Gamma \vdash e : \tau$ **exp**.

**Proof:** By induction on the definition of the derived forms. The proof is straightforward: I give here one example case in detail.

1. $\Psi;\Delta;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2$ **dexp** By inversion (lemma 16, $\Psi;\Delta;\Gamma \vdash e_1 : \tau_1$ **exp** and $\Psi;\Delta;\Gamma \vdash e_2 : \tau_2$ **exp**. There are three possible cases in the definition of the derived form.

   (a) $e_1 = sv_1, e_2 = sv_2$. Then $\langle sv_1, sv_2\rangle \stackrel{\text{def}}{=} \mathtt{let}\, x = \langle sv_1, sv_2\rangle\, \mathtt{in}\, x$, which is well-formed by construction using the assumptions and the freshness of $x$.

   (b) $e_1 = sv_1, e_2 = \mathtt{let}\, x = i\, \mathtt{in}\, e_2'$. Then $\langle sv_1, e_2\rangle \stackrel{\text{def}}{=} \mathtt{let}\, x = i\, \mathtt{in}\, \langle sv_1, e_2'\rangle$. By inverting the second assumption (3), $\Psi;\Delta;\Gamma \vdash i : \tau_x$ $\mathbf{opr}_{32}$ and $\Psi;\Delta;\Gamma, x{:}\tau_x \vdash e_2' : \tau_2$ **exp**. By weakening the first assumption (lemma 2), $\Psi;\Delta;\Gamma, x{:}\tau_x \vdash sv_1 : \tau_1$ **exp**. By the derived typing rule for pairs $\Psi;\Delta;\Gamma, x{:}\tau_x \vdash \langle sv_1, e_2'\rangle : \tau_1 \times \tau_2$ **dexp**, and hence by induction $\Psi;\Delta;\Gamma, x{:}\tau_x \vdash \langle sv_1, e_2'\rangle : \tau_1 \times \tau_2$ **exp**. Finally, by construction using the operation rule, $\Psi;\Delta;\Gamma \vdash \mathtt{let}\, x = i\, \mathtt{in}\, \langle sv_1, e_2'\rangle : \tau_1 \times \tau_2$ **exp**.

(c) $e_1 = \mathtt{let}\, x = i\, \mathtt{in}\, e_1', e_2 = sv_2$. This case proceeds exactly as the previous case.

2. The other cases proceed similarly.

■

Finally, I show the soundness of the dynamic encoding translation, using the soundness theorems for the **SEK**, **SE**, and **DET** translations, along with the commutativity lemma for the **DET** translation and the soundness lemmas for contexts.

**Theorem 3 (Soundness of the dynamic encoding translation)**
*If* $\Delta \vdash c : \kappa$ *and* $\vdash \Psi$ **ok** *then* $\Psi; |\Delta|; \downharpoonleft\Delta\downharpoonright \vdash \downharpoonleft c \downharpoonright \,:\, \downharpoonleft c{:}\kappa \downharpoonright$ **exp**.

**Proof:** By induction on derivations of $\Delta \vdash c : \kappa$.

1. $\Delta[\alpha{:}\kappa] \vdash \alpha : \kappa$.

   By assumption:
   $\vdash \Delta[\alpha{:}\kappa]$ **ok**
   $\vdash \Psi$ **ok**

   By lemmas 7 and 15:
   $\vdash |\Delta[\alpha{:}\kappa]|$ **ok**
   $|\Delta[\alpha{:}\kappa]| \vdash \downharpoonleft\Delta[\alpha{:}\kappa]\downharpoonright$ **ok**

   By definition:
   $\downharpoonleft\alpha\downharpoonright \stackrel{\text{def}}{=} x_\alpha$
   $\downharpoonleft\Delta[\alpha{:}\kappa]\downharpoonright \stackrel{\text{def}}{=} \downharpoonleft\Delta\downharpoonright\, [x_\alpha{:}\, \downharpoonleft\alpha{:}\kappa\downharpoonright]$

   Hence by the variable rule:
   $\Psi; |\Delta[\alpha{:}\kappa]|; \downharpoonleft\Delta\downharpoonright\, [x_\alpha{:}\, \downharpoonleft\alpha{:}\kappa\downharpoonright] \vdash x_\alpha :\, \downharpoonleft\alpha{:}\kappa\downharpoonright$ **exp**

2. $\Delta \vdash \lambda(\alpha{:}\kappa_1).c : \kappa_1 \to \kappa_2$.

   By assumption:
   $\vdash \kappa_1$ **ok**, $\alpha \notin fv(\Delta)$
   $\Delta, \alpha{:}\kappa_1 \vdash c : \kappa_2$

   By theorem 1 and lemma 12:
   $|\Delta, \alpha{:}\kappa_1| \vdash |\kappa|$ **ok**
   $|\Delta, \alpha{:}\kappa_1| \vdash \downharpoonleft\alpha{:}\kappa_1\downharpoonright\, : \mathrm{T}_{32}$

   By induction:
   $\Psi; |\Delta, \alpha{:}\kappa_1|; \downharpoonleft\Delta, \alpha{:}\kappa\downharpoonright \vdash \downharpoonleft c\downharpoonright \,:\, \downharpoonleft c{:}\kappa_2\downharpoonright$ **exp**

   So by the function introduction rule:
   $\Psi; |\Delta|; \downharpoonleft\Delta\downharpoonright \vdash \downharpoonleft\lambda\alpha{:}\kappa_1.c\downharpoonright \,:\, \downharpoonleft\lambda\alpha{:}\kappa_1.c{:}\kappa_1 \to \kappa_2\downharpoonright$ **exp**

3. $\Delta \vdash c_1 c_2 : \kappa_2$.

   By assumption:
   $\Delta \vdash c_1 : \kappa_1 \to \kappa_2$
   $\Delta \vdash c_2 : \kappa_1$

By induction:
$\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \lfloor c_1 \rfloor \,:\, \lfloor c_1{:}\kappa_1 \to \kappa_2 \rfloor$ **exp**
$\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \lfloor c_2 \rfloor \,:\, \lfloor c_2{:}\kappa_1 \rfloor$ **exp**

By theorem 2:
$|\Delta| \vdash |c_2| : |\kappa_1|$

Note that $\lfloor c_1{:}\kappa_1 \to \kappa_2 \rfloor \stackrel{\text{def}}{=} \forall[\alpha{:}|\kappa_1|](\lfloor \alpha{:}\kappa_1 \rfloor) \to \lfloor c_1 \alpha{:}\kappa_2 \rfloor$.

By lemma 13:
$\lfloor \alpha{:}\kappa_1 \rfloor\ [|c_2|/\alpha] = \lfloor c_2{:}\kappa_1 \rfloor$
$\lfloor c_1 \alpha{:}\kappa_2 \rfloor\ [|c_2|/\alpha] = \lfloor c_1 c_2{:}\kappa_2 \rfloor$

(Note $\alpha \notin fv(c_1)$ by freshness).

Therefore, by the derived rule for application of expressions:
$\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash c_1 c_2 : \lfloor c_1 c_2{:}\kappa_2 \rfloor$ **dexp**

Finally, observe that by lemma 17:
$\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash c_1 c_2 : \lfloor c_1 c_2{:}\kappa_2 \rfloor$ **exp**

4. $\Delta \vdash \pi_1 c : \kappa_1$.

   By assumption:
   $\Delta \vdash c : \kappa_1 \times \kappa_2$

   By induction
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \lfloor c \rfloor \,:\, \lfloor c{:}\kappa_1 \times \kappa_2 \rfloor$ **exp**

   By definition:
   $\lfloor c{:}\kappa_1 \times \kappa_2 \rfloor = \lfloor \pi_1 c{:}\kappa_1 \rfloor \times \lfloor \pi_2 c{:}\kappa_2 \rfloor$

   By the typing rule for the $\texttt{select}^i\, e$ derived form:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \texttt{select}^1\ \lfloor c \rfloor \,:\, \lfloor \pi_1 c{:}\kappa_1 \rfloor$ **dexp**

   By lemma 17:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \texttt{select}^1\ \lfloor c \rfloor \,:\, \lfloor \pi_1 c{:}\kappa_1 \rfloor$ **exp**

5. $\Delta \vdash \pi_2 c : \kappa_2$. The proof proceeds as in the previous case.

6. $\Delta \vdash \langle c_1, c_2 \rangle : \kappa_1 \times \kappa_2$

   By assumption:
   $\Delta \vdash c_1 : \kappa_1$
   $\Delta \vdash c_2 : \kappa_2$

   By induction
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \lfloor c_1 \rfloor \,:\, \lfloor c_1{:}\kappa_1 \rfloor$ **exp**
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \lfloor c_2 \rfloor \,:\, \lfloor c_2{:}\kappa_2 \rfloor$ **exp**

   By the derived pair introduction rule:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \langle \lfloor c_1 \rfloor, \lfloor c_2 \rfloor \rangle : \lfloor c_1{:}\kappa_1 \rfloor \times \lfloor c_2{:}\kappa_2 \rfloor$ **dexp**

   By theorem 17:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \langle \lfloor c_1 \rfloor, \lfloor c_2 \rfloor \rangle : \lfloor c_1{:}\kappa_1 \rfloor \times \lfloor c_2{:}\kappa_2 \rfloor$ **exp**

   By the definitions of the translations and the beta rule for pairs:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor \vdash \lfloor \langle c_1, c_2 \rangle \rfloor \,:\, \lfloor \langle c_1, c_2 \rangle{:}\kappa_1 \times \kappa_2 \rfloor$ **exp**

7. The rest of the cases follow directly by construction, with appeals to previous lemmas where necessary.

■

## 5.4 Type translations

### 5.4.1 Proper MIL types

The **DE** and **SE** translations complete the apparatus needed to translate proper **MIL** constructors. Constructors of kind $T_{32}$ map to constructors of kind $T_{\mathrm{mil}}$, 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 $T_{\mathrm{mil}}$ kind, but instead goes directly to kind $T_{32}$.

$$
\begin{array}{lll}
\multicolumn{2}{l}{|\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n](\tau_1,\ldots,\tau_m)(k)\to\tau|} & \stackrel{\mathrm{def}}{=} \\
\multicolumn{3}{l}{\quad\forall[\alpha_1{:}|\kappa_1|,\ldots,\alpha_n{:}|\kappa_n|]} \\
\multicolumn{3}{l}{\qquad(\downharpoonleft\alpha_1{:}\kappa_1\downharpoonright,\ldots\downharpoonleft\alpha_n{:}\kappa_n\downharpoonright,|\tau_1|,\ldots,|\tau_m|)(\mathtt{Float}_0,\ldots,\mathtt{Float}_{k-1})\to|\tau|} \\
|T(c)| & \stackrel{\mathrm{def}}{=} & \text{interp}\ |c| \\
|\tau_1\times\ldots\times\tau_n| & \stackrel{\mathrm{def}}{=} & \times[|\tau_1|,\ldots,|\tau_n|]
\end{array}
$$

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.

### 5.4.2 The term typing context translation

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

$$
\begin{array}{lll}
|\bullet| & \stackrel{\mathrm{def}}{=} & \bullet \\
|\Gamma,x{:}\tau| & \stackrel{\mathrm{def}}{=} & |\Gamma|,x{:}|\tau| \\
|\Gamma,x_{64}| & \stackrel{\mathrm{def}}{=} & |\Gamma|,x_{64}{:}\mathtt{Float}
\end{array}
$$

### 5.4.3 Proofs of soundness for the proper type translations

**Theorem 4 (Soundness of the type translation)**
*If* $\Delta\vdash\tau$ **ok** *then* $|\Delta|\vdash|\tau|:T_{32}$

**Proof:** By induction on $\tau$. We proceed by cases:

- $\Delta\vdash T(c)$ **ok**

  By assumption:
  $\Delta\vdash c:T_{32}$

By theorem 2:
$|\Delta| \vdash |c| : |\mathrm{T}_{32}|$

Note that $|\mathrm{T}_{32}| = \mathrm{T}_{\mathrm{mil}}$

By lemma 8:
$|\Delta| \vdash \mathrm{interp} : \mathrm{T}_{\mathrm{mil}} \to \mathrm{T}_{32}$

By the application rule:
$|\Delta| \vdash \mathrm{interp}(|c|) : \mathrm{T}_{32}$

- $\Delta \vdash \tau_1 \times \ldots \times \tau_n$ **ok**

  By assumption:
  $\Delta \vdash \tau_1\ \mathbf{ok} \ldots \Delta \vdash \tau_n\ \mathbf{ok}$

  By induction:
  $|\Delta| \vdash |\tau_1| : \mathrm{T}_{32} \ldots |\Delta| \vdash |\tau_n| : \mathrm{T}_{32}$

  By construction:
  $|\Delta| \vdash \times[|\tau_1|, \ldots, |\tau_n|]\ \mathbf{ok}\mathrm{T}_{32}$

- $\Delta \vdash \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau$ **ok**

  By assumption:
  $\Delta \vdash \kappa_i\ \mathbf{ok} \quad i \in 1 \ldots n$
  $\Delta[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n] \vdash \tau_i\ \mathbf{ok} \quad i \in 1 \ldots m$
  $\Delta[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n] \vdash \tau\ \mathbf{ok}$

  By theorem 1 and lemma 12:
  $|\Delta| \vdash \kappa_i\ \mathbf{ok} \quad i \in 1 \ldots n$
  $|\Delta[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]| \vdash \lfloor \alpha_i{:}\kappa_i \rfloor : \mathrm{T}_{32} \quad i \in 1 \ldots n$

  By induction:
  $|\Delta[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]| \vdash |\tau_i| : \mathrm{T}_{32} \quad i \in 1 \ldots m$
  $|\Delta[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]| \vdash |\tau| : \mathrm{T}_{32}$

  By the function type introduction rule:
  $$|\Delta| \vdash \left\{ \begin{array}{l} \forall[\alpha_1{:}|\kappa_1|, \ldots, \alpha_n{:}|\kappa_n|] \\ \quad (\lfloor \alpha_1{:}\kappa_1 \rfloor, \ldots, \lfloor \alpha_n{:}\kappa_n \rfloor, |\tau_1|, \ldots, |\tau_m|)(\texttt{Float}_1 \ldots \texttt{Float}_k) \to |\tau| \end{array} \right\} : \mathrm{T}_{32}$$

∎

**Lemma 18**

1. *If $x \notin \Gamma$ then $x \notin |\Gamma|$.*

2. *If $x_{64} \notin \Gamma$ then $x_{64} \notin |\Gamma|$.*

**Proof:** Note that $\Gamma$ and $|\Gamma|$ have the same domain by construction.

∎

**Lemma 19 (Term context translation)**
*If $\Delta \vdash \Gamma$ **ok** then $|\Delta| \vdash |\Gamma|$ **ok**.*

**Proof:** By induction on derivations, using the domain lemma (lemma 18), the soundness lemma for the constructor translation (lemma 7) and the soundness theorem for the type translation (theorem 4).

∎

**Corollary 2**
*If* $\Delta \vdash \Gamma$ **ok** *then* $|\Delta| \vdash \lrcorner\Delta\llcorner, |\Gamma|$ **ok**.

**Proof:** By induction on $\Gamma$, using lemmas 18, 19 and 15. Note that we assume that indexed variables $x_\alpha$ are drawn from a "fresh" set, and hence there is no interference between the domains of $\lrcorner\Delta\llcorner$ and $|\Gamma|$.

∎

### 5.4.4 The type translation respects equivalence

**Lemma 20 (The type translation respects equivalence)**
*If* $\Delta \vdash \tau_1 \equiv \tau_2 : \mathrm{T}_{32}$*, then* $|\Delta| \vdash |\tau_1| \equiv |\tau_2| : \mathrm{T}_{32}$.

**Proof:** By induction on equivalence derivations, appealing to lemma 10 and the structural equivalence rule for application in the base case ($T(c)$).

∎

## 5.5 The term translation

The translation from **MIL** terms to **LIL** terms is not much more complicated than the constructor translation, but is syntactically somewhat more cumbersome. In order to ensure that the appropriate type information is available for constructing **LIL** terms, it is convenient to phrase the translations on terms as typed translations of the form $\Delta; \Gamma \vdash e : \tau \leadsto e'$ indicating that in the typing context $\Delta; \Gamma$, a **MIL** term $e$ of type $\tau$ translates to a **LIL** term $e'$.

### 5.5.1 Definitions for the term translation

In order to more concisely state the translation itself, I first define a number of **LIL** functions implementing type analysis primitives. In a practical implementation, it may be desirable to inline some or all of these functions: for the most part however this is purely a policy decision trading off code size against function calls. Note though that in general it is not always possible to eliminate the lambda abstraction since the type analysis mechanism requires that certain types be a variable. In the case that the type to be analyzed is an application of a variable, the lambda abstraction cannot be eliminated.

The first defined form is the optimized array constructor, essentially as described in the example from chapter 4. I again use ML style pattern matching at the term level for clarity of presentation.

Note that the definitions are closed and hence can be written directly as code.

$$
\begin{array}{l}
\textbf{array} \stackrel{\text{def}}{=} \\
\quad \texttt{code}[\alpha{:}\mathrm{T_{mil}}](\lambda(r_\alpha{:}\,R(\alpha), i{:}\texttt{Int}, v{:}\,\mathrm{interp}(\alpha))(){:}\texttt{Array}(\alpha). \\
\qquad \texttt{case}\, r_\alpha \\
\qquad\quad \textbf{Record}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{Record}\, \_ \Rightarrow \texttt{array}_{\mathrm{interp}(\alpha)}(i, v) \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad\quad \mid \textbf{BFloat}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{BFloat}\, \_ \Rightarrow \texttt{array}_{\texttt{Float}}(i, \texttt{unbox}\, v)) \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad\quad \mid \textbf{Other}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{Other}\, \_ \Rightarrow \texttt{array}_{\mathrm{interp}(\alpha)}(i, v)) \mid \_ \Rightarrow \texttt{dead}\, x)
\end{array}
$$

Here the **DE** argument, $r_\alpha$ is the dynamic encoding of the constructor represented by $\alpha$. As can be seen from the definition of $R$, the type of $r_\alpha$ is therefore a term-level sum describing the top level structure of the encoded type. The appropriate branch 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.

In addition to the array creation function, I define specialized update and subscript functions for the optimized arrays.

$$
\begin{array}{l}
\textbf{upd} \stackrel{\text{def}}{=} \\
\quad \texttt{code}[\alpha{:}\mathrm{T_{mil}}](r_\alpha{:}\,R(\alpha), a{:}\,\textbf{Array}(\alpha), i{:}\texttt{Int}, v{:}\,\mathrm{interp}(\alpha))(){:}\texttt{Unit}. \\
\qquad \texttt{case}\, r_\alpha \\
\qquad\quad \textbf{Record}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{Record}\, \_ \Rightarrow \texttt{upd}_{\mathrm{interp}(\alpha)}(a, i, v) \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad\quad \mid \textbf{BFloat}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{BFloat}\, \_ \Rightarrow \texttt{upd}_{\texttt{Float}}(a, i, \texttt{unbox}\, v)) \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad\quad \mid \textbf{Other}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{Other}\, \_ \Rightarrow \texttt{upd}_{\mathrm{interp}(\alpha)}(a, i, v)) \mid \_ \Rightarrow \texttt{dead}\, x)
\end{array}
$$

$$
\begin{array}{l}
\textbf{sub} \stackrel{\text{def}}{=} \\
\quad \texttt{code}[\alpha{:}\mathrm{T_{mil}}](r_\alpha{:}\,R(\alpha), a{:}\,\textbf{Array}(\alpha), i{:}\texttt{Int})(){:}\,\mathrm{interp}(\alpha). \\
\qquad \texttt{case}\, r_\alpha \\
\qquad\quad \textbf{Record}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{Record}\, \_ \Rightarrow \texttt{sub}_{\mathrm{interp}(\alpha)}(a, i) \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad\quad \mid \textbf{BFloat}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{BFloat}\, \_ \Rightarrow \texttt{box}(\texttt{sub}_{\texttt{Float}}(a, i)) \mid \_ \Rightarrow \texttt{dead}\, x) \\
\qquad\quad \mid \textbf{Other}\, x \Rightarrow \texttt{vcase}\, \alpha\, (\textbf{Other}\, \_ \Rightarrow \texttt{sub}_{\mathrm{interp}(\alpha)}(a, i)) \mid \_ \Rightarrow \texttt{dead}\, x)
\end{array}
$$

Notice that all of the array primitives, in the case that the array is a flattened float array, will perform boxing or unboxing. If the type is statically known, both the branching and the boxing can be optimized away, but in general they cannot be avoided. This makes it very important to recognize statically reducible uses of type dispatch.

The `vararg` and `onearg` primitives from the **MIL** correspond in a similar way to the defined **vararg** and **onearg** functions in the **LIL**. Notice that as with the **Vararg** type function, I choose to make **vararg** polymorphic over the encoding of the argument type, but the *interpretation* of the encoded return type (that is, the return type itself).

$$
\begin{array}{l}
\textbf{vararg} \stackrel{\text{def}}{=} \\
\quad \texttt{code}[\alpha{:}\mathrm{T}_{\mathrm{mil}}, \rho{:}\mathrm{T}_{32}](r_\alpha{:}\, R(\alpha), f{:}(\mathrm{interp}(\alpha) \rightarrow \rho))()\colon \textbf{Vararg}(\alpha)(\rho). \\
\quad\quad \texttt{case}\, r_\alpha( \\
\quad\quad\quad \textbf{Record}\, y \Rightarrow \texttt{vcase}\, \alpha\, ( \\
\quad\quad\quad\quad \textbf{Record}\, \alpha' \Rightarrow \texttt{case}\, y \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_0\, x \Rightarrow \texttt{vcase}\, \alpha' \\
\quad\quad\quad\quad\quad\quad ([] \Rightarrow \lambda().(f\langle\rangle) \\
\quad\quad\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_1\, x \Rightarrow \texttt{vcase}\, \alpha' \\
\quad\quad\quad\quad\quad\quad ([\beta] \Rightarrow \lambda(y{:}\beta).(f\langle y\rangle) \\
\quad\quad\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_2\, x \Rightarrow \texttt{vcase}\, \alpha' \\
\quad\quad\quad\quad\quad\quad ([\beta_1, \beta_2] \Rightarrow \lambda(y_1{:}\beta_1, y_2{:}\beta_2).(f\langle y_1, y_2\rangle) \\
\quad\quad\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_3\, \_ \Rightarrow f \\
\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, y) \\
\quad\quad\quad \mid \_ \Rightarrow f)
\end{array}
$$

The implementations of **vararg** and **onearg** are more complicated than those of the array primitives. In addition to dispatching on the top level form of the type of the argument, they 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.

$$
\begin{array}{l}
\textbf{onearg} \stackrel{\text{def}}{=} \\
\quad \texttt{code}[\alpha{:}\mathrm{T}_{\mathrm{mil}}, \rho{:}\mathrm{T}_{32}]\lambda(r_\alpha{:}\, R(\alpha), f{:}\, \textbf{Vararg}(\alpha)(\rho))(){:}(\mathrm{interp}(\alpha) \rightarrow \rho). \\
\quad\quad \texttt{case}\, r_\alpha( \\
\quad\quad\quad \textbf{Record}\, y \Rightarrow \texttt{vcase}\, \alpha\, ( \\
\quad\quad\quad\quad \textbf{Record}\, \alpha' \Rightarrow \texttt{case}\, y \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_0\, x \Rightarrow \texttt{vcase}\, \alpha' \\
\quad\quad\quad\quad\quad\quad ([] \Rightarrow \lambda(y{:}\texttt{unit}).(f()) \\
\quad\quad\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_1\, x \Rightarrow \texttt{vcase}\, \alpha' \\
\quad\quad\quad\quad\quad\quad ([\beta] \Rightarrow \lambda(\langle y\rangle{:} \times [\beta]).(f(y)) \\
\quad\quad\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_2\, x \Rightarrow \texttt{vcase}\, \alpha' \\
\quad\quad\quad\quad\quad\quad ([\beta_1, \beta_2] \Rightarrow \lambda(\langle y_1, y_2\rangle{:} \times [\beta_1, \beta_2]).(f(y_1, y_2)) \\
\quad\quad\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, x) \\
\quad\quad\quad\quad\quad \mid \texttt{inj}_3\, \_ \Rightarrow f \\
\quad\quad\quad\quad \mid \_ \Rightarrow \texttt{dead}\, y) \\
\quad\quad\quad \mid \_ \Rightarrow f)
\end{array}
$$

I assign a **LIL** heap label to each of the named definitions: for example, vararg is the label given to the **vararg** definition.

### 5.5.2 The term level translation

The complete **MIL** to **LIL** term translation is given in section 5.6, but a selection of example rules from the translation are discussed below to illustrate the important points of the translation.

#### Small values

The small value translation judgement is of the form $\Delta; \Gamma \vdash sv : \tau \rightsquigarrow sv'$, and is for the most part uncomplicated.

Injections into non-value carrying arms of sum types are translated into uses of union types, as described previously. Note that injections into value-carrying arms are not values in the **MIL** since they require allocation. Their translation is therefore described in the operation translation.

$$\frac{}{\Delta; \Gamma \vdash \mathtt{inj\_tag}^{j}_{\mathtt{Sum}_i(\vec{c})} : \mathtt{Sum}^{j}_{i}(\vec{c}) \rightsquigarrow \mathtt{inj\_union}_{|\,\mathtt{Sum}_i(\vec{c})|}\ \mathtt{tag}_j}$$

#### Float values

The float value translation judgement is of the form $\Delta; \Gamma \vdash fv : \mathtt{Float} \rightsquigarrow fv'$. Float values in the **MIL** and the **LIL** correspond exactly: hence the translation leaves the terms unchanged.

#### 64 bit operations

The 64 bit operation translation judgement is of the form $\Delta \vdash i : \mathtt{Float} \rightsquigarrow i'$, and is also relatively uncomplicated. For example, the floating point subscript operation changes in only minor syntactic ways.

$$\frac{\Delta; \Gamma \vdash sv_1 : \mathtt{Farray} \rightsquigarrow sv_1' \quad \Delta; \Gamma \vdash sv_2 : \mathtt{Int} \rightsquigarrow sv_2'}{\Delta; \Gamma \vdash \mathtt{fsub}(sv_1, sv_2) : \mathtt{Float} \rightsquigarrow \mathtt{sub}_{\mathtt{Float}}(sv_1', sv_2')}$$

#### 32 bit operations

The 32 bit operation translation judgement is of the form $\Delta; \Gamma \vdash i : \tau \rightsquigarrow i'$. Much of the interesting work of the translation takes place in the operations.

Although the most significant part of the **MIL** to **LIL** translation is the representation of type analysis within the term language, the definitions given in section 5.5.1 isolate all of the uses of type analysis. Consequently, the translation of type analyzing primitives such as `vararg` is very straightforward: they simply become calls to the code definitions via the appropriate heap labels.

$$\frac{\Delta; \Gamma \vdash sv : c_1 \to c_2 \rightsquigarrow sv'}{\Delta; \Gamma \vdash \mathtt{vararg}_{c_1 \to c_2}\, sv : \mathtt{Vararg}_{c_1 \to c_2} \rightsquigarrow \mathtt{call}\, \mathsf{vararg}[|c_1|, \mathrm{interp}\, |c_2|](\lfloor c_1 \rfloor, sv')}$$

Notice that as previously discussed, I pass the interpretation of the encoded return type to the **vararg** code, instead of the encoded type itself. The dynamic encoding of the argument type is passed as an additional argument to the function to allow the dispatch on types to take place.

Sum types become union types, with the tagging of records made explicit.

$$\frac{\Delta;\Gamma \vdash sv : c_j \leadsto sv'}{\Delta;\Gamma \vdash \mathtt{inj}^{j}_{\mathtt{Sum}_i(\vec{c})}\ sv : \mathtt{Sum}^{j}_{i}(\vec{c}) \leadsto \mathtt{inj\_union}_{|\,\mathtt{Sum}_i(\vec{c})|}\langle \mathtt{tag}_j, sv'\rangle}$$

As discussed in the constructor translation, the known sum type from the **MIL** disappears in the **LIL**. Uses of the projection from this type become second projections out of a tag-value pair.

$$\frac{\Delta;\Gamma \vdash sv : \mathtt{Sum}^{j}_{i}(\vec{c}) \leadsto sv'}{\Delta;\Gamma \vdash \mathtt{proj}^{j}_{\mathtt{Sum}_i(\vec{c})}\ sv : c_j \leadsto \mathtt{select}^2\ sv'}$$

In the **LIL** the exception type is no longer primitive, instead being replaced by uses of existentials and pairs.

$$\frac{\Delta;\Gamma \vdash sv_1 : \mathtt{Dyntag}_c \leadsto sv_1' \quad \Delta;\Gamma \vdash sv_2 : c \leadsto sv_2'}{\begin{array}{l}\Delta;\Gamma \vdash \mathtt{inj\_dyn}_c(sv_1, sv_2) : \mathtt{Exn} \leadsto \\ \quad \mathtt{pack}\langle sv_1', sv_2'\rangle\ \mathtt{as}\ \exists\alpha{:}\mathrm{T}_{32}.(\mathtt{Dyntag}(\alpha) \times \alpha)\ \mathtt{hiding}\ |c|\end{array}}$$

Since the exception type is closed, I will frequently refer to it via the following definition for brevity:

$$\mathbf{Dyn} \stackrel{\text{def}}{=} \exists\alpha{:}\mathrm{T}_{32}.(\mathtt{Dyntag}(\alpha) \times \alpha)$$

One of the most illustrative rules is the rule for the translation of applications, since it makes clear the "plumbing" work necessary to make type analysis explicit.

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv : \forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau \leadsto sv' \\ \Delta;\Gamma \vdash sv_i : \tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n] \leadsto sv_i' \quad \Delta;\Gamma \vdash fv_i : \mathtt{Float} \leadsto fv_i'\end{array}}{\begin{array}{r}\Delta;\Gamma \vdash sv[c_1, \ldots, c_n](sv_1, \ldots, sv_m)(fv_1, \ldots, fv_k) : \tau[c_1/\alpha_1, \ldots, c_n/\alpha_n] \leadsto \\ sv'[|c_1|, \ldots, |c_n|](\rfloor c_1 \lfloor, \ldots, \rfloor c_n \lfloor, sv_1', \ldots, sv_m')(fv_1', \ldots, fv_k')\end{array}}$$

Notice that each type argument is passed to the function once as a type argument in its static encoding, and once as a term argument in its dynamic encoding. The rest of the arguments are translated using the small value and float value translations and are passed as usual. The application translation will of course be paralleled exactly by the translation of functions in the expression translation below.

### The term translation judgements

The term level translation judgement is of the form $\Delta;\Gamma \vdash e : \tau \leadsto e'$. For the most part, the expression translation simply appeals to the various other judgements. The only interesting case is that of polymorphic functions.

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

As expected from the application case, each type argument is both lifted to the static encoding kind and used to construct a dynamic encoding type for a new term level argument serving as its dynamic encoding.

**Programs**

The final **LIL** program is produced from a closed **MIL** expression by wrapping the result of the expression translation with a heap binding the defined forms used in the translation.

$$
\begin{array}{lllll}
d_i \stackrel{\mathrm{def}}{=} & \bullet, & & \\
& \mathsf{sub} & :\forall[\alpha{:}\mathrm{T_{mil}}]\,\mathtt{Code}[R(\alpha), \mathbf{Array}(\alpha), \mathtt{Int}][](\mathrm{interp}(\alpha)) & \mapsto \mathbf{sub}, \\
& \mathsf{upd} & :\forall[\alpha{:}\mathrm{T_{mil}}]\,\mathtt{Code}[R(\alpha), \mathbf{Array}(\alpha), \mathtt{Int}, \mathrm{interp}(\alpha)][](\mathtt{unit}) & \mapsto \mathbf{upd}, \\
& \mathsf{array} & :\forall[\alpha{:}\mathrm{T_{mil}}]\,\mathtt{Code}[R(\alpha), \mathtt{Int}, \mathrm{interp}(\alpha)][](\mathtt{Array}(\alpha)) & \mapsto \mathbf{array}, \\
& \mathsf{vararg} & :\forall[\alpha{:}\mathrm{T_{mil}}, \rho{:}\mathrm{T_{32}}]\,\mathtt{Code}[R(\alpha), (\mathrm{interp}(\alpha) \to \rho)][](\mathbf{Vararg}(\alpha)(\rho)) & \mapsto \mathbf{vararg}, \\
& \mathsf{onearg} & :\forall[\alpha{:}\mathrm{T_{mil}}, \rho{:}\mathrm{T_{32}}]\,\mathtt{Code}[R(\alpha), \mathbf{Vararg}(\alpha)(\rho)][](\mathrm{interp}(\alpha) \to \rho) & \mapsto \mathbf{onearg}
\end{array}
$$

$$
\frac{\bullet;\bullet \vdash e : \tau \rightsquigarrow e'}{\vdash e : \tau \rightsquigarrow \mathtt{letrec}\; d_i \;\mathtt{in}\; e' \;\mathbf{prog}}
$$

### 5.5.3 Proof that the term level translation preserves typing

**Small Values**

**Theorem 5 (Small value translation)**
*If* $\Delta;\Gamma \vdash sv : \tau$, $\vdash \Psi$ **ok** *and* $\Delta;\Gamma \vdash sv : \tau \rightsquigarrow sv'$ *then* $\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |\tau|$

**Proof:** By induction on terms.

1. If $sv = x$ then

   By assumption:
   $\Delta{:}\Gamma, x{:}\tau \vdash x : \tau$
   $\vdash \Psi$ **ok**

   By inverting the assumption:
   $\Delta \vdash \Gamma, x{:}\tau$ **ok**

   By lemma 19:
   $|\Delta| \vdash \lfloor\Delta\rfloor, |\Gamma, x{:}\tau|$ **ok**

   By the variable rule and the definition of $|\Gamma|$:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma|, x{:}|\tau| \vdash x : |\tau|$

2. If $sv = i$ then

   By assumption:
   $\Delta{:}\Gamma \vdash i : \mathtt{Int}$
   $\vdash \Psi$ **ok**

   By inverting the assumption:
   $\Delta \vdash \Gamma$ **ok**

By lemma 19:
$|\Delta| \vdash \lfloor\Delta\rfloor, |\Gamma|$ **ok**

By the int rule:
$\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash i : \texttt{Int}$

3. If $sv = \texttt{unroll}_c\, sv$ then

By assumption:
$\Delta; \Gamma \vdash sv : \pi_i\, \mu(\alpha, \beta)(c_1, c_2)$
$\Delta \vdash c \equiv \pi_i\, \mu(\alpha, \beta)(c_1, c_2) : \mathrm{T}_{32}$
$\Delta \vdash sv : \pi_i\, \mu(\alpha, \beta)(c_1, c_2) \rightsquigarrow sv'$

By induction:
$\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |\,\pi_i\, \mu(\alpha, \beta)(c_1, c_2)|$

By lemma 10:
$|\Delta| \vdash |c| \equiv |\,\pi_i\, \mu(\alpha, \beta)(c_1, c_2)| : \mathrm{T}_{\mathrm{mil}}$

By definition:
$|\,\pi_i\, \mu(\alpha, \beta)(c_1, c_2)| = \texttt{Rec}[1+1](f)(\texttt{inj}_i\, *)$
where $f = \lambda(\rho{:}1+1 \to \mathrm{T}_{32}).\lambda(\omega.1+1).$
$\texttt{case}\, \omega$
$\quad \texttt{inj}_1\, \_ \Rightarrow \mathrm{interp}(|c_1|[\mathbf{Other}(\rho(\texttt{inj}_1\, *))/\alpha, \mathbf{Other}(\rho(\texttt{inj}_2\, *))/\beta])$
$\quad |\, \texttt{inj}_2\, \_ \Rightarrow \mathrm{interp}(|c_2|[\mathbf{Other}(\rho(\texttt{inj}_1\, *))/\alpha, \mathbf{Other}(\rho(\texttt{inj}_2\, *))/\beta])$

Therefore, by the unroll rule:
$\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{unroll}_{|c|}\, sv' : f(\texttt{Rec}[1+1](f))(\texttt{inj}_i\, *)$

It suffices to show that:
$f(\texttt{Rec}[1+1](f))(\texttt{inj}_i\, *) = \mathrm{interp}(|c_i[\pi_1\, \mu(\alpha, \beta)(c_1, c_2)/\alpha, \pi_2\, \mu(\alpha, \beta)(c_1, c_2)/\beta]|)$

By lemma 9:
$|c_i[\pi_1\, \mu(\alpha, \beta)(c_1, c_2)/\alpha, \pi_2\, \mu(\alpha, \beta)(c_1, c_2)/\beta]| =$
$|c_i|[|\,\pi_1\, \mu(\alpha, \beta)(c_1, c_2)|/\alpha, |\,\pi_2\, \mu(\alpha, \beta)(c_1, c_2)|/\beta]$

By definition:
$|\,\pi_1\, \mu(\alpha, \beta)(c_1, c_2)| = \mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_1\, *))$
$|\,\pi_2\, \mu(\alpha, \beta)(c_1, c_2)| = \mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_2\, *))$

So it suffices to show that:
$f(\texttt{Rec}[1+1](f))(\texttt{inj}_i\, *) =$
$\quad \mathrm{interp}(|c_i|[\mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_1\, *))/\alpha, \mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_2\, *))/\beta])$

By definition of $f$:
$f(\texttt{Rec}[1+1](f))(\texttt{inj}_i\, *) =$
$\texttt{case}\ (\texttt{inj}_i\, *)$
$\quad \texttt{inj}_1\, \_ \Rightarrow$
$\quad\quad \mathrm{interp}(|c_1|[\mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_1\, *))/\alpha, \mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_2\, *))/\beta])$
$\quad |\, \texttt{inj}_2\, \_ \Rightarrow$
$\quad\quad \mathrm{interp}(|c_2|[\mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_1\, *))/\alpha, \mathbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_2\, *))/\beta])$

If $i = 1$, then by the beta rule:
$f(\texttt{Rec}[1+1](f))(\texttt{inj}_1 *) =$
$\text{interp}(|c_1|[\textbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_1 *))/\alpha, \textbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_2 *))/\beta])$

If $i = 2$, then by the beta rule:
$f(\texttt{Rec}[1+1](f))(\texttt{inj}_2 *) =$
$\text{interp}(|c_2|[\textbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_1 *))/\alpha, \textbf{Other}(\texttt{Rec}[1+1](f)(\texttt{inj}_2 *))/\beta])$

4. The `roll` case follows similarly.

5. If $sv = \texttt{inj\_tag}^{j}_{\texttt{Sum}_i(\vec{c})}$ then:

   By inversion of the first assumption:
   $\Delta \vdash \Gamma$ **ok**

   By lemma 19:
   $|\Delta| \vdash \lfloor\Delta\rfloor, |\Gamma|$ **ok**

   Therefore by the tag rule:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{tag}_j : \texttt{Tag}(j)$

   Note that
   $|\,\texttt{Sum}_i(c_i, \ldots, c_n)| =$
   $\bigvee[\texttt{Tag}(0), \ldots, \texttt{Tag}(i-1), \ldots, \times[\texttt{Tag}(i), \text{interp}\,|c_i|], \ldots, \times[\texttt{Tag}(n), \text{interp}\,|c_n|]]$

   So by the union formation rule :
   $\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{inj\_union}_{|\,\texttt{Sum}_i(\vec{c})|}\, \texttt{tag}_j : |\,\texttt{Sum}_i(\vec{c})|$

■

**Float values**

**Theorem 6 (Float value translation)**
*If* $\Delta; \Gamma \vdash fv : \texttt{Float}$, $\vdash \Psi$ **ok** *and* $\Delta; \Gamma \vdash fv : \texttt{Float} \rightsquigarrow fv'$ *then* $|\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash fv' : \texttt{Float}$

**Proof:**

1. If $fv = x_f$ then

   By assumption:
   $\Delta{:}\Gamma, x_f \vdash x : \texttt{Float}$
   $\vdash \Psi$ **ok**

   By inverting the assumption:
   $\Delta \vdash \Gamma, x_f$ **ok**

   By lemma 19:
   $|\Delta| \vdash \lfloor\Delta\rfloor, |\Gamma, x_f|$ **ok**

   By the variable rule and the definition of $|\Gamma|$:
   $\Psi; |\Delta|; \lfloor\Delta\rfloor, |\Gamma|, x_f \vdash x_f : \texttt{Float}$

2. $fv = r$.

   By assumption:
   $\Delta{:}\Gamma \vdash r : \texttt{Float}$
   $\vdash \Psi$ **ok**

By inverting the assumption:
$\Delta \vdash \Gamma$ **ok**

By lemma 19:
$|\Delta| \vdash \lfloor \Delta \rfloor, |\Gamma|$ **ok**

By the float rule:
$\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash r : \texttt{Float}$

∎

**64 bit operations**

**Theorem 7 (Float operation translation)**
*If* $\Delta; \Gamma \vdash i : \texttt{Float}$, $\vdash \Psi$ **ok**, *and* $\Delta; \Gamma \vdash i : \texttt{Float} \rightsquigarrow i'$ *then* $|\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash i' : \texttt{Float}\ \mathbf{opr}_{64}$

**Proof:**

1. If $i = fv$ then:

   By inversion of assumptions:
   $\Delta; \Gamma \vdash fv : \texttt{Float}$
   $\Delta; \Gamma \vdash fv : \texttt{Float} \rightsquigarrow fv'$

   By theorem 6:
   $\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash fv' : \texttt{Float}$

   By the float value inclusion rule:
   $\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash fv' : \texttt{Float}\ \mathbf{opr}_{64}$

2. If $i = \texttt{unboxf}\ sv$ then:
   By inversion of assumptions:
   $\Delta; \Gamma \vdash sv : \texttt{Boxf}$
   $\Delta; \Gamma \vdash sv : \texttt{Boxf} \rightsquigarrow sv'$

   By theorem 5:
   $\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash sv' : \texttt{Boxed}\,\texttt{Float}$

   By the unbox rule:
   $\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash \texttt{unbox}\, sv' : \texttt{Float}\ \mathbf{opr}_{64}$

3. If $i = \texttt{fsub}(sv_1, sv_2)$ then:

   By inversion of assumptions:
   $\Delta; \Gamma \vdash sv_1 : \texttt{Farray}$
   $\Delta; \Gamma \vdash sv_2 : \texttt{Int}$
   $\Delta; \Gamma \vdash sv_1 : \texttt{Farray} \rightsquigarrow sv_1'$
   $\Delta; \Gamma \vdash sv_2 : \texttt{Int} \rightsquigarrow sv_2'$

   By theorem 5:
   $\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash sv_1' : \texttt{Array}_{64}\texttt{Float}$
   $\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash sv_2' : \texttt{Int}$

   Hence by the 64 bit subscript rule :
   $\Psi; |\Delta|; \lfloor \Delta \rfloor, |\Gamma| \vdash \texttt{sub}_{\texttt{Float}}(sv_1', sv_2') : \texttt{Float}\ \mathbf{opr}_{64}$

∎

### Operations and expressions

I begin with a lemma to capture the appropriate typing conditions for the definitions used by the translation of the type analyzing primitives.

**Lemma 21 (Definitions)**

1. *If* $\vdash \Psi$ **ok** *then*

$$\Psi \vdash \mathbf{sub} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\,\mathtt{Code}[R(\alpha), \mathbf{Array}(\alpha), \mathtt{Int}][](\mathrm{interp}(\alpha))\ \mathbf{hval}$$

2. *If* $\vdash \Psi$ **ok** *then*

$$\Psi \vdash \mathbf{upd} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\,\mathtt{Code}[R(\alpha), \mathbf{Array}(\alpha), \mathtt{Int}, \mathrm{interp}(\alpha)][](\mathtt{unit})\ \mathbf{hval}$$

3. *If* $\vdash \Psi$ **ok** *then*

$$\Psi \vdash \mathbf{array} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\,\mathtt{Code}[R(\alpha), \mathtt{Int}, \mathrm{interp}(\alpha)][](\mathbf{Array}(\alpha))\ \mathbf{hval}$$

4. *If* $\vdash \Psi$ **ok** *then*

$$\Psi \vdash \mathbf{vararg} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}, \beta{:}\mathrm{T}_{32}]\,\mathtt{Code}[R(\alpha), \mathrm{interp}(\alpha) \to \beta)][](\mathbf{Vararg}(\alpha)(\beta))\ \mathbf{hval}$$

5. *If* $\vdash \Psi$ **ok** *then*

$$\Psi \vdash \mathbf{onearg} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}, \beta{:}\mathrm{T}_{32}]\,\mathtt{Code}[R(\alpha), \mathbf{Vararg}(\alpha)(\beta)][](\mathrm{interp}(\alpha) \to \beta)\ \mathbf{hval}$$

**Proof:** By construction.

■

I define a heap context $\Psi_i$ that gives types for the above definitions as follows:

$$\begin{array}{lll}
\Psi_i \stackrel{\mathrm{def}}{=} & \bullet, & \\
& \mathsf{sub} & :\forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\,\mathtt{Code}[R(\alpha), \mathbf{Array}(\alpha), \mathtt{Int}][](\mathrm{interp}(\alpha)), \\
& \mathsf{upd} & :\forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\,\mathtt{Code}[R(\alpha), \mathbf{Array}(\alpha), \mathtt{Int}, \mathrm{interp}(\alpha)][](\mathtt{unit}), \\
& \mathsf{array} & :\forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\,\mathtt{Code}[R(\alpha), \mathtt{Int}, \mathrm{interp}(\alpha)][](\mathtt{Array}(\alpha)), \\
& \mathsf{vararg} & :\forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}, \rho{:}\mathrm{T}_{32}]\,\mathtt{Code}[R(\alpha), (\mathrm{interp}(\alpha) \to \rho)][](\mathbf{Vararg}(\alpha)(\rho)), \\
& \mathsf{onearg} & :\forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}, \rho{:}\mathrm{T}_{32}]\,\mathtt{Code}[R(\alpha), \mathbf{Vararg}(\alpha)(\rho)][](\mathrm{interp}(\alpha) \to \rho)
\end{array}$$

**Lemma 22 (Initial context)**
$\vdash \Psi_i$ **ok**

**Proof:** By construction.

■

Operations and expressions are mutually recursive: hence their soundness theorems are stated most naturally together. Since the translation of operations generates calls to the code functions implementing type analysis, the soundness theorem for the translation is defined relative to the initial context defined above.

**Theorem 8 (Operation and expression translation)**

1. *If* $\Delta; \Gamma \vdash i : \tau$ *and* $\Delta; \Gamma \vdash i : \tau \rightsquigarrow i'$ *then* $\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash i' : |\tau|\ \mathbf{opr}_{32}$

2. *If* $\Delta;\Gamma \vdash e:\tau$ *and* $\Delta;\Gamma \vdash e:\tau \rightsquigarrow e'$ *then* $\Psi_i;|\Delta|;\lfloor\Delta\rfloor,|\Gamma| \vdash e':|\tau|\ \mathbf{exp}$

**Proof:**

The proof proceeds by simultaneous induction on operations and expressions. We begin with the operations.

1. If $i = sv$ then:
   By inversion of assumptions:
   $\Delta;\Gamma \vdash sv:\tau$
   $\Delta;\Gamma \vdash sv:\tau \rightsquigarrow sv'$

   By theorem 5:
   $\Psi_i;|\Delta|;\lfloor\Delta\rfloor,|\Gamma| \vdash sv':|\tau|$

   By the small value expression inclusion rule:
   $\Psi_i;|\Delta|;\lfloor\Delta\rfloor,|\Gamma| \vdash sv':|\tau|\ \mathbf{opr}_{32}$

2. If $i = \mathtt{vararg}_{c_1\to c_2}\ sv$
   By inversion of assumptions:
   $\Delta \vdash c_1:\mathrm{T}_{32}$
   $\Delta \vdash c_2:\mathrm{T}_{32}$
   $\Delta;\Gamma \vdash sv:c_1 \to c_2$
   $\Delta;\Gamma \vdash sv:c_1 \to c_2 \rightsquigarrow sv'$

   By theorem 2:
   $|\Delta| \vdash |c_1|:\mathrm{T}_{\mathrm{mil}}$
   $|\Delta| \vdash |c_2|:\mathrm{T}_{\mathrm{mil}}$

   By lemma 8:
   $|\Delta| \vdash \mathrm{interp}\,|c_2|:\mathrm{T}_{32}$

   By theorem 3 and weakening:
   $\Psi_i;|\Delta|;\lfloor\Delta\rfloor,|\Gamma| \vdash \lfloor c_1\rfloor\ :\ R(c_1)$ (Note $\lfloor c_1{:}\mathrm{T}_{32}\rfloor = R(c_1)$).

   By theorem 5:
   $\Psi_i;|\Delta|;\lfloor\Delta\rfloor,|\Gamma| \vdash sv':|c_1 \to c_2|$

   By the initial context lemma (22):
   $\Psi_i;|\Delta|;\lfloor\Delta\rfloor,|\Gamma| \vdash \mathsf{onearg}:\forall[\alpha{:}\mathrm{T}_{\mathrm{mil}},\rho{:}\mathrm{T}_{32}]\,\mathtt{Code}[R(\alpha),\mathrm{interp}(\alpha)\to\beta][](\mathbf{Vararg}(\alpha)(\rho))$

   By the code call rule:
   $\Psi_i;|\Delta|;\lfloor\Delta\rfloor,|\Gamma| \vdash \mathtt{call}\,\mathsf{vararg}[|c_1|,\mathrm{interp}\,|c_2|](\lfloor c_1\rfloor,sv'):\ \mathbf{Vararg}(|c_1|)(\mathrm{interp}\,|c_2|)\ \mathbf{opr}_{32}$

3. If $i = \mathtt{onearg}_{c_1\to c_2}\ sv$
   By inversion of assumptions:
   $\Delta \vdash c_1:\mathrm{T}_{32}$
   $\Delta \vdash c_2:\mathrm{T}_{32}$
   $\Delta;\Gamma \vdash sv:\mathtt{Vararg}_{c_1\to c_2}$
   $\Delta;\Gamma \vdash sv:\mathtt{Vararg}_{c_1\to c_2} \rightsquigarrow sv'$

   By theorem 2:
   $|\Delta| \vdash |c_1|:\mathrm{T}_{\mathrm{mil}}$
   $|\Delta| \vdash |c_2|:\mathrm{T}_{\mathrm{mil}}$

By lemma 8:
$|\Delta| \vdash \mathrm{interp}\,|c_2| : \mathrm{T}_{32}$

By theorem 3 and weakening:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \lfloor c_1 \rfloor : R(|c_1|)$ (Note $\lfloor c_1{:}\mathrm{T}_{32}\rfloor = R(|c_1|)$).

By theorem 5:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : \mathbf{Vararg}(|c_1|)(\mathrm{interp}\,|c_2|)$
(Note $|\mathtt{Vararg}_{c_1 \to c_2}| = \mathbf{Vararg}(|c_1|)(\mathrm{interp}\,|c_2|)$)

By the initial context lemma (22):
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathsf{vararg} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}, \rho{:}\mathrm{T}_{32}]\,\mathtt{Code}[R(\alpha), \mathbf{Vararg}(\alpha)(\rho)][](\mathrm{interp}(\alpha) \to \beta)$

By the code call rule:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathtt{call}\,\mathtt{onearg}[|c_1|, \mathrm{interp}\,|c_2|](\lfloor c_1 \rfloor, sv') : \mathrm{interp}\,|c_1| \to \mathrm{interp}\,|c_2|\ \mathbf{opr}_{32}$

4. If $i = \mathtt{boxf}\,\mathit{fv}$ then

   By inversion of assumptions:
   $\Delta; \Gamma \vdash \mathit{fv} : \mathtt{Float}$
   $\Delta; \Gamma \vdash \mathit{fv} : \mathtt{Float} \rightsquigarrow \mathit{fv}'$

   By theorem 6:
   $\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathit{fv}' : \mathtt{Float}$

   By the box rule:
   $\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathtt{box}\,\mathit{fv}' : \mathtt{Boxed}\,\mathtt{Float}$

5. If $i = \mathtt{proj}^j_{\mathtt{Sum}_i(\vec{c})}\,sv$ then

   By inversion of assumptions:
   $\Delta \vdash \mathtt{Sum}^j_i(\vec{c}) : \mathrm{T}_{32}$
   $\Delta; \Gamma \vdash sv : \mathtt{Sum}^j_i(c_i, \ldots, c_j, \ldots, c_n)$
   $\Delta; \Gamma \vdash sv : \mathtt{Sum}^j_i(c_i, \ldots, c_j, \ldots, c_n) \rightsquigarrow sv'$

   By theorem 5:
   $\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |\,\mathtt{Sum}^j_i(c_i, \ldots, c_j, \ldots, c_n)|$

   By definition:
   $|\,\mathtt{Sum}^j_i(c_1, \ldots, c_j, \ldots, c_n)| = \times[\mathtt{Tag}(j), \mathrm{interp}\,|c_j|]$

   Hence by the selection rule:
   $\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathtt{select}^2\,sv' : \mathrm{interp}\,|c_j|$
   (Note the implicit type inclusion, hence $\mathrm{interp}\,|c_j| = |T(c_j)|$)

6. If $i = \mathtt{inj}^j_{\mathtt{Sum}_i(c_i, \ldots, c_j, \ldots, c_n)}\,sv$ then

   By inverting the assumptions:
   $\Delta \vdash \mathtt{Sum}_i(c_i, \ldots, c_j, \ldots, c_n) : \mathrm{T}_{32}$
   $\Delta; \Gamma \vdash sv : c_j$
   $\Delta; \Gamma \vdash sv : c_j \rightsquigarrow sv'$

   By theorem 5:
   $\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |c_j|$

By the pair and tag rules:
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \langle \mathtt{tag}_j, sv' \rangle :\ \times[\mathtt{Tag}(j), |c_j|]$

By definition:
$|\,\mathtt{Sum}_i(c_i, \ldots, c_n)| = \bigvee[\mathtt{Tag}(0), \ldots, \mathtt{Tag}(i-1), \ldots, \times[\mathtt{Tag}(i), |c_i|], \ldots, \times[\mathtt{Tag}(n), |c_n|]]$

By the union injection rule:
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \mathtt{inj\_union}_{|\,\mathtt{Sum}_i(\vec{c})|} \langle \mathtt{tag}_j, sv' \rangle : |\,\mathtt{Sum}_i(\vec{c})|$

7. If $i = \langle sv'_1, \ldots, sv'_n \rangle$ then:

   By inverting assumptions:
   $\Delta; \Gamma \vdash sv_i : \tau_i \quad i \in 1 \ldots n$
   $\Delta; \Gamma \vdash sv_i : \tau_i \rightsquigarrow sv'_i \quad i \in 1 \ldots n$

   By theorem 5:
   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash sv'_i : |\tau_i| \quad i \in 1 \ldots n$

   By the tuple rule:
   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \langle sv'_1, \ldots, sv'_n \rangle :\ \times[|\tau_1|, \ldots, |\tau_n|]$

8. If $i = \mathtt{select}^i\, sv$ then:

   By inverting assumptions:
   $\Delta; \Gamma \vdash sv : \tau_1 \times \ldots \times \tau_n$
   $\Delta; \Gamma \vdash sv : \tau_1 \times \ldots \times \tau_n \rightsquigarrow sv'$

   By theorem 5:
   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash sv' : |\tau_1 \times \ldots \times \tau_n|$

   By the select rule:
   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \mathtt{select}^i\, sv' : |\tau_i|$

9. If $i = \mathtt{case}_\tau(sv)\,(x_1.e_1, \ldots, x_n.e_n)$ then:

   By inverting assumptions:
   $\Delta \vdash \tau : \mathrm{T}_{32}$
   $\Delta; \Gamma \vdash sv : \mathtt{Sum}_i(\vec{c})$
   $\Delta; \Gamma[x_j : \mathtt{Sum}_i^j(\vec{c})] \vdash e_j : \tau$
   $\Delta; \Gamma \vdash sv : \mathtt{Sum}_i(\vec{c}) \rightsquigarrow sv'$
   $\Delta; \Gamma[x_j : \mathtt{Sum}_i^j(\vec{c})] \vdash e_j : \tau \rightsquigarrow e'_j$

   By theorem 4:
   $|\Delta| \vdash |\tau| : \mathrm{T}_{32}$

   By theorem 5:
   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash sv' : |\,\mathtt{Sum}_i(\vec{c})|$

   By induction:
   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma[x_j : \mathtt{Sum}_i^j(\vec{c})]| \vdash e'_j : |\tau|\ \mathbf{exp}$

   Note that:
   $|\Gamma[x_j : \mathtt{Sum}_i^j(\vec{c})]| = |\Gamma|[x_j : |\,\mathtt{Sum}_i^j(\vec{c})|]$
   $|\,\mathtt{Sum}_i(c_i, \ldots, c_n)| = \bigvee[\mathtt{Tag}(0), \ldots, \mathtt{Tag}(i-1), \ldots, \times[\mathtt{Tag}(i), |c_i|], \ldots, \times[\mathtt{Tag}(n), |c_n|]]$
   $|\,\mathtt{Sum}_i^j(c_1, \ldots, c_n)| = \mathtt{Tag}(j) \quad j < i$
   $|\,\mathtt{Sum}_i^j(c_1, \ldots, c_n)| = \times[\mathtt{Tag}(j), c_j] \quad i \leq j \leq n$

Hence by the case introduction rule:
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \texttt{case}_{|\tau|}(sv')\,(x_1.e'_1, \ldots, x_n.e'_n) : |\tau|$ **exp**

10. If $i = \texttt{handle}_\tau(e_1, x.e_2)$ then:

By inverting assumptions:
$\Delta \vdash \tau : \mathrm{T}_{32}$
$\Delta; \Gamma \vdash e_1 : \tau$
$\Delta; \Gamma[x : \texttt{Exn}] \vdash e_2 : \tau$
$\Delta; \Gamma \vdash e_1 : \tau \rightsquigarrow e'_1$
$\Delta; \Gamma[x : \texttt{Exn}] \vdash e_2 : \tau \rightsquigarrow e'_2$

By theorem 4:
$|\Delta| \vdash |\tau| : \mathrm{T}_{32}$

By induction:
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash e'_1 : |\tau|$ **exp**
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma[x{:}\texttt{Exn}]| \vdash e'_2 : |\tau|$ **exp**

By definition:
$|\Gamma[x{:}\texttt{Exn}]| = |\Gamma|[x{:}\,\texttt{Dyn}]$

So by the handle rule :
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \texttt{handle}_{|\tau|}(e'_1, x.e'_2) : |\tau|\ \mathbf{opr}_{32}$

11. If $i = sv[c_1, \ldots, c_n](sv_1, \ldots, sv_m)(fv_1, \ldots, fv_k)$ then:

By inverting assumptions:
$\Delta; \Gamma \vdash sv : \forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau$
$\Delta; \Gamma \vdash sv_i : \tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n]$
$\Delta; \Gamma \vdash fv_i : \texttt{Float}$
$\Delta; \Gamma \vdash sv : \forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau \rightsquigarrow sv'$
$\Delta; \Gamma \vdash sv_i : \tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n] \rightsquigarrow sv'_i$
$\Delta; \Gamma \vdash fv_i : \texttt{Float} \rightsquigarrow fv'_i$

By theorems 5 and 6:
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash sv' : |\forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau|$
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash sv'_i : |\tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n]|$
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash fv'_i : \texttt{Float}$

By definition:
$|\forall[\alpha{::}\kappa_1, \ldots, \alpha{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau| =$
$\forall[\alpha_1{::}|\kappa_1|, \ldots, \alpha_n{::}|\kappa_n|](\downharpoonleft\alpha_1{:}\kappa_1\downharpoonright, \ldots, \downharpoonleft\alpha_n{:}\kappa_n\downharpoonright, |\tau_1|, \ldots, |\tau_m|)(\texttt{Float}_0 \ldots \texttt{Float}_k) \to |\tau|$

By theorem 2:
$|\Delta| \vdash |c_i| : |\kappa| \quad i \in 1 \ldots n$

Therefore by the type instantiation rule :
$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash sv'[|c_1|, \ldots, |c_n|] :$
$((\downharpoonleft\alpha_1{:}\kappa_1\downharpoonright, \ldots, \downharpoonleft\alpha_n{:}\kappa_n\downharpoonright, |\tau_1|, \ldots, |\tau_m|)(\texttt{Float}_0 \ldots \texttt{Float}_k) \to |\tau|)[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n]$

To apply the function application rule, is suffices to show that:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \lfloor c_i \rfloor \;:\; \lfloor \alpha_i{:}\kappa_i \rfloor\; [|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] \quad i \in 1 \ldots n$
and
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_i : |\tau_i|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] \quad i \in 1 \ldots m$

By theorem 3:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \lfloor c_i \rfloor \;:\; \lfloor c_i{:}\kappa_i \rfloor \quad i \in 1 \ldots n$

By lemma 13:
$\lfloor \alpha_i{:}\kappa_i \rfloor\; [|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] = \lfloor \alpha_i[c_1/\alpha_1, \ldots, c_n/\alpha_n]{:}\kappa_i \rfloor = \lfloor c_i{:}\kappa_i \rfloor$

We know from above that:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_i : |\tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n]| \quad i \in 1 \ldots m$

But by lemma 9:
$|\tau_i|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] = |\tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n]|$

Therefore, by the application rule :
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'[|c_1|, \ldots, |c_n|](\lfloor c_1 \rfloor, \ldots, \lfloor c_n \rfloor, sv'_1, \ldots, sv'_m)(fv'_1, \ldots, fv'_k) :$
$|\tau|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n]$

Finally, note that by lemma 9:
$|\tau|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] = |\tau[c_1/\alpha_1, \ldots, c_n/\alpha_n]|$

12. If $i = \texttt{raise}_\tau\, sv$ then:

By inverting assumptions:
$\Delta \vdash \tau : \mathrm{T}_{32}$
$\Delta; \Gamma \vdash sv : \texttt{Exn}$
$\Delta; \Gamma \vdash sv : \texttt{Exn} \rightsquigarrow sv'$

By theorem 4:
$|\Delta| \vdash |\tau| : \mathrm{T}_{32}$

By theorem 5:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |\texttt{Exn}|$

Hence by the raise introduction rule:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{raise}_{|\tau|}\, sv' : |\tau|$

13. If $i = \texttt{mkexntag}_c$ then:

By inverting assumptions:
$\Delta \vdash c : \mathrm{T}_{32}$
$\Delta \vdash \Gamma$ **ok**

By theorem 4:
$|\Delta| \vdash |c| : \mathrm{T}_{32}$

Hence by the dyntag intro rule:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{dyntag}_{|c|} : \texttt{Dyntag}(|c|)$

14. If $i = \texttt{exncase}_\tau(sv)\,(sv_1 \Rightarrow x_1.e_1, \_ \Rightarrow e_2)$ then:

By inverting assumptions:
$\Delta; \Gamma \vdash sv : \mathtt{Exn}$
$\Delta \vdash \tau : \mathrm{T}_{32}$
$\Delta; \Gamma \vdash sv_1 : \mathtt{Dyntag}_{c_1}$
$\Delta; \Gamma[x_1{:}c_1] \vdash e_1 : \tau$
$\Delta; \Gamma \vdash e_2 : \tau$
$\Delta; \Gamma \vdash sv : \mathtt{Exn} \leadsto sv'$
$\Delta; \Gamma \vdash sv_1 : \mathtt{Dyntag}_{c_1} \leadsto sv'_1$
$\Delta; \Gamma[x_1{:}c_1] \vdash e_1 : \tau \leadsto e'_1$
$\Delta; \Gamma \vdash e_2 : \tau \leadsto e'_2$

By theorem 4:
$|\Delta| \vdash |\tau| : \mathrm{T}_{32}$

By theorem 5:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |\mathtt{Exn}|$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_1 : |\,\mathtt{Dyntag}_{c_1}\,|$

By induction:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma[x_1{:}c_1]| \vdash e'_1 : |\tau|\ \mathbf{exp}$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash e'_2 : |\tau|\ \mathbf{exp}$

Hence by the dyncase introduction rule:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathtt{dyncase}_\tau(sv)\,(sv'_1 \Rightarrow x_1.e'_1, \_ \Rightarrow e'_2) : |\tau|\ \mathbf{opr}_{32}$

15. If $i = \mathtt{sub}_c(sv_1, sv_2)$ then:

By inverting assumptions:
$\Delta \vdash c : \mathrm{T}_{32}$
$\Delta; \Gamma \vdash sv_1 : \mathtt{Array}_c$
$\Delta; \Gamma \vdash sv_2 : \mathtt{Int}$
$\Delta; \Gamma \vdash sv_1 : \mathtt{Array}_c \leadsto sv'_1$
$\Delta; \Gamma \vdash sv_2 : \mathtt{Int} \leadsto sv'_2$

By theorem 4:
$|\Delta| \vdash |c| : \mathrm{T}_{32}$

By theorem 5:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_1 : |\mathtt{Array}_c|$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_2 : |\mathtt{Int}|$

By theorem 3:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \lfloor c\rfloor\ : R(|c|)$

By lemma 22:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathtt{sub} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\,\mathtt{Code}[R(\alpha), \mathbf{Array}(\alpha), \mathtt{Int})][](\mathrm{interp}(\alpha))$

Hence by the code call rule :
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathtt{call}\,\mathtt{sub}[|c|](\lfloor c\rfloor, sv'_1, sv'_2) : \mathrm{interp}\,|c|\ \mathbf{opr}_{32}$

16. If $i = \mathtt{upd}_c(sv_1, sv_2, sv_3)$ then:

By inverting assumptions:
$\Delta \vdash c : \mathrm{T}_{32}$
$\Delta; \Gamma \vdash sv_1 : \texttt{Array}_c$
$\Delta; \Gamma \vdash sv_2 : \texttt{Int}$
$\Delta; \Gamma \vdash sv_3 : c$
$\Delta; \Gamma \vdash sv_1 : \texttt{Array}_c \rightsquigarrow sv'_1$
$\Delta; \Gamma \vdash sv_2 : \texttt{Int} \rightsquigarrow sv'_2$
$\Delta; \Gamma \vdash sv_3 : c \rightsquigarrow sv'_3$

By theorem 4:
$|\Delta| \vdash |c| : \mathrm{T}_{32}$

By theorem 5:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_1 : |\texttt{Array}_c|$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_2 : |\texttt{Int}|$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_3 : |c|$

By theorem 3:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \lfloor c\rfloor : R(|c|)\ \mathbf{exp}$

By lemma 22:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathbf{upd} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\, \texttt{Code}[R(\alpha), \mathbf{Array}(\alpha), \texttt{Int}, \mathrm{interp}(\alpha)][](\texttt{unit})$

Hence by the code call rule :
$|\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{call}\,\mathsf{upd}[|c|](\lfloor c\rfloor, sv'_1, sv'_2, sv'_3) : \texttt{Unit}\ \mathbf{opr}_{32}$

17. If $i = \texttt{array}_c(sv_1, sv_2)$ then:

By inverting assumptions:
$\Delta \vdash c : \mathrm{T}_{32}$
$\Delta; \Gamma \vdash sv_1 : \texttt{Int}$
$\Delta; \Gamma \vdash sv_2 : c$
$\Delta; \Gamma \vdash sv_1 : \texttt{Int} \rightsquigarrow sv'_2$
$\Delta; \Gamma \vdash sv_2 : c \rightsquigarrow sv'_2$

By theorem 4:
$|\Delta| \vdash |c| : \mathrm{T}_{32}$

By theorem 5:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_1 : |\texttt{Int}|$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv'_2 : |c|$

By theorem 3:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \lfloor c\rfloor : R(|c|)$

By lemma 22:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \mathsf{array} : \forall[\alpha{:}\mathrm{T}_{\mathrm{mil}}]\, \texttt{Code}[R(\alpha), \texttt{Int}, \mathrm{interp}(\alpha)][](\mathbf{Array}(\alpha))$

Hence by the code call rule :
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{call}\,\mathsf{array}[|c|](\lfloor c\rfloor, sv'_1, sv'_2) : \texttt{Array}(\mathrm{interp}\,|c|)\ \mathbf{opr}_{32}$

18. If $i = \texttt{fupd}(sv_1, sv_2, fv)$ then:

By inverting assumptions:
$\Delta;\Gamma \vdash sv_1 : \texttt{Farray}$
$\Delta;\Gamma \vdash sv_2 : \texttt{Int}$
$\Delta;\Gamma \vdash fv :$
$\Delta;\Gamma \vdash sv_1 : \texttt{Farray} \rightsquigarrow sv_1'$
$\Delta;\Gamma \vdash sv_2 : \texttt{Int} \rightsquigarrow sv_2'$
$\Delta;\Gamma \vdash fv : \texttt{Float} \rightsquigarrow fv'$

By theorems 5 and 6:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv_1' : |\texttt{Farray}|$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv_2' : \texttt{Int}$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash fv' : \texttt{Float}$

By the 64 bit array update rule:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{upd}_{\texttt{Float}}(sv_1', sv_2', fv') : \texttt{Unit}\ \mathbf{opr}_{32}$

19. If $i = \texttt{array}(sv, fv)$ then:

By inverting assumptions:
$\Delta;\Gamma \vdash sv : \texttt{Int}$
$\Delta;\Gamma \vdash fv :$
$\Delta;\Gamma \vdash sv : \texttt{Int} \rightsquigarrow sv'$
$\Delta;\Gamma \vdash fv : \texttt{Float} \rightsquigarrow fv'$

By theorems 5 and 6:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : \texttt{Int}$
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash fv' : \texttt{Float}$

By the 64 bit array creation rule:
$|\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash \texttt{upd}_{\texttt{Float}}(sv', fv') : \texttt{Farray}(\texttt{Float})\ \mathbf{opr}_{32}$

For expressions

1. If $e = sv$ then:

By inversion of assumptions:
$\Delta;\Gamma \vdash sv : \tau$
$\Delta;\Gamma \vdash sv : \tau \rightsquigarrow sv'$

By theorem 5:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |\tau|$

By the small value expression inclusion rule:
$\Psi_i; |\Delta|; \lfloor\Delta\rfloor, |\Gamma| \vdash sv' : |\tau|\ \mathbf{exp}$

2. If $e = \texttt{let}_\tau\, x = i\, \texttt{in}\, e$ then:

By inversion of assumptions:
$\Delta;\Gamma \vdash i : \tau_i$
$\Delta;\Gamma, x{:}\tau_i \vdash e : \tau$
$\Delta;\Gamma \vdash i : \tau_i \rightsquigarrow i'$
$\Delta;\Gamma, x{:}\tau_i \vdash e : \tau \rightsquigarrow e'$

By induction:

$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash i' : |\tau_i|\ \mathbf{opr}_{32}$

$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma|, x{:}|\tau_i| \vdash e' : |\tau|\ \mathbf{exp}$ (Note that $|\Gamma, x{:}\tau_i| = |\Gamma|, x{:}|\tau_i|$)

By the operation rule:

$\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \mathtt{let}_{|\tau|}\, x = i'\, \mathtt{in}\, e' : |\tau|\ \mathbf{exp}$

3. If $e = \mathtt{let}_\tau\, x_f = i\, \mathtt{in}\, e$ then:

   By inversion of assumptions:

   $\Delta; \Gamma \vdash i : \mathtt{Float}$

   $\Delta; \Gamma, x_f \vdash e : \tau$

   $\Delta; \Gamma \vdash i : \mathtt{Float} \rightsquigarrow i'$

   $\Delta; \Gamma, x_f \vdash e : \tau \rightsquigarrow e'$

   By induction and theorem 7:

   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash i' : \mathtt{Float}\ \mathbf{opr}_{64}$

   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma|, x_f{:}\mathtt{Float} \vdash e' : |\tau|\ \mathbf{exp}$ (Note that $|\Gamma, x_f| = |\Gamma|, x_f{:}\mathtt{Float}$)

   By the 64 bit operation rule:

   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash \mathtt{let}_{|\tau|}\, x_f = i'\, \mathtt{in}\, e' : |\tau|\ \mathbf{exp}$

4. if $e = \mathtt{let}_\tau\, \mathtt{rec}_{\tau_r}\, f[\overrightarrow{\alpha::\kappa}](\overrightarrow{x{:}\tau})(\vec{x_f}).e_f\, \mathtt{in}\, e$ then:

   By inversion of assumptions:

   $\Delta, \overrightarrow{\alpha::\kappa}; \Gamma[f : \forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r, \overrightarrow{x{:}\tau}, \vec{x_f}] \vdash e_f : \tau_r$

   $\Delta; \Gamma[f : \forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r] \vdash e : \tau$

   $\Delta, \overrightarrow{\alpha::\kappa}; \Gamma[f : \forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r, \overrightarrow{x{:}\tau}, \vec{x_f}] \vdash e_f : \tau_r \rightsquigarrow e'_f$

   $\Delta; \Gamma[f : \forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r] \vdash e : \tau \rightsquigarrow e'$

   By induction:

   $\Psi_i; |\Delta, \overrightarrow{\alpha::\kappa}|; \downharpoonleft\Delta, \overrightarrow{\alpha::\kappa}\downharpoonright, |\Gamma[f : \forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r, \overrightarrow{x{:}\tau}, \vec{x_f}]| \vdash e'_f : |\tau_r|\ \mathbf{exp}$

   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma[f : \forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r]| \vdash e' : |\tau|\ \mathbf{exp}$

   By definition:

   $\downharpoonleft\Delta, \overrightarrow{\alpha::\kappa}\downharpoonright = \downharpoonleft\Delta\downharpoonright, x_\alpha{:}\ \overrightarrow{\downharpoonleft\alpha{:}\kappa\downharpoonright}$

   And

   By definition:

   $|\Gamma[f : \forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r, \overrightarrow{x{:}\tau}, \vec{x_f}]| =$

   $\quad |\Gamma|[f : |\forall[\overrightarrow{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r|, x{:}\overrightarrow{|\tau|}, x_f{:}\overrightarrow{\mathtt{Float}}] =$

   $\quad |\Gamma|[f : \forall[\alpha{::}\overrightarrow{|\kappa|}](\overrightarrow{\downharpoonleft\alpha{:}\kappa\downharpoonright}, \overrightarrow{|\tau|})(\overrightarrow{\mathtt{Float}}) \to |\tau_r|, x{:}\overrightarrow{|\tau|}, \overrightarrow{x_f{:}\mathtt{Float}}]$

   Hence by the rec rule:

   $\Psi_i; |\Delta|; \downharpoonleft\Delta\downharpoonright, |\Gamma| \vdash$

   $\mathtt{let}_{|\tau|}\, \mathtt{rec}_{|\tau_r|}\, f[\overrightarrow{\alpha::|\kappa|}](\overrightarrow{x_\alpha{:}\ \downharpoonleft\alpha::\kappa\downharpoonright}, \overrightarrow{x{:}\tau})(\overrightarrow{x_f{:}\mathtt{Float}}).e'_f\, \mathtt{in}\, e' : |\tau|$

∎

**Programs**

The correctness of the program translation follows almost immediately.

**Theorem 9 (Programs)**
*If* $\bullet;\bullet \vdash e:$ *and* $\vdash e : \tau \rightsquigarrow \texttt{let}\, d_i \,\texttt{in}\, e'$ **prog** *then* $\bullet \vdash \texttt{let}\, d_i \,\texttt{in}\, e : |\tau|$.

**Proof:** (By construction)
By lemma 21 and the heap well-formedness rule:
$\Psi_i \vdash d_i$ **ok**
By theorem 8
$\Psi_i;\bullet;\bullet \vdash e' : |\tau|$ **exp**
So by the program well-formedness rule
$\bullet \vdash \texttt{let}\, d_i \,\texttt{in}\, e : |\tau|$

■

## 5.6 The complete term level translation

**Small Values** $\boxed{\Delta;\Gamma \vdash sv : \tau \rightsquigarrow sv'}$

$$\frac{}{\Delta;\Gamma[x{:}\tau] \vdash x : \tau \rightsquigarrow x}$$

$$\frac{}{\Delta;\Gamma \vdash i : \texttt{Int} \rightsquigarrow i}$$

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

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \pi_i\, \mu(\alpha,\beta)(c_1,c_2) : \mathrm{T}_{32} \\ \Delta;\Gamma \vdash sv : c_i[\pi_1\, \mu(\alpha,\beta)(c_1,c_2)/\alpha, \pi_2\, \mu(\alpha,\beta)(c_1,c_2)/\beta] \rightsquigarrow sv'\end{array}}{\begin{array}{r}\Delta;\Gamma \vdash \texttt{roll}_c\, sv : \pi_i\, \mu(\alpha,\beta)(c_1,c_2) \rightsquigarrow \\ \texttt{roll}_{|c|}\, sv'\end{array}}$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{inj\_tag}^{j}_{\texttt{Sum}_i(\vec{c})} : \texttt{Sum}^{j}_{i}(\vec{c}) \rightsquigarrow \texttt{inj\_union}_{|\,\texttt{Sum}_i(\vec{c})|}\, \texttt{tag}_j}$$

**Float values** $\boxed{\Delta \vdash fv : \texttt{Float} \rightsquigarrow fv'}$

$$\frac{}{\Delta;\Gamma \vdash r : \texttt{Float} \rightsquigarrow r}$$

$$\frac{}{\Delta;\Gamma[x_f] \vdash x_f : \texttt{Float} \rightsquigarrow x_f}$$

**64 bit operations** $\boxed{\Delta \vdash i : \texttt{Float} \rightsquigarrow i'}$

$$\frac{\Delta; \Gamma \vdash \mathit{fv} : \texttt{Float} \rightsquigarrow \mathit{fv}'}{\Delta; \Gamma \vdash \mathit{fv} : \texttt{Float} \rightsquigarrow \mathit{fv}'}$$

$$\frac{\Delta; \Gamma \vdash sv : \texttt{Boxf} \rightsquigarrow sv'}{\Delta; \Gamma \vdash \texttt{unboxf}\, sv : \texttt{Float} \rightsquigarrow \texttt{unbox}\, sv'}$$

$$\frac{\Delta; \Gamma \vdash sv_1 : \texttt{Farray} \rightsquigarrow sv_1' \quad \Delta; \Gamma \vdash sv_2 : \texttt{Int} \rightsquigarrow sv_2'}{\Delta; \Gamma \vdash \texttt{fsub}(sv_1, sv_2) : \texttt{Float} \rightsquigarrow \texttt{sub}_{\texttt{Float}}(sv_1', sv_2')}$$

**Operations** $\boxed{\Delta; \Gamma \vdash i : \tau \rightsquigarrow i'}$

$$\frac{\Delta; \Gamma \vdash sv : \tau \rightsquigarrow sv'}{\Delta; \Gamma \vdash sv : \tau \rightsquigarrow sv'}$$

$$\frac{\Delta; \Gamma \vdash sv : c_1 \to c_2 \rightsquigarrow sv'}{\Delta; \Gamma \vdash \texttt{vararg}_{c_1 \to c_2}\, sv : \texttt{Vararg}_{c_1 \to c_2} \rightsquigarrow \texttt{call}\, \mathsf{vararg}[|c_1|, \mathrm{interp}\, |c_2|](\downharpoonleft c_1 \downharpoonright, sv')}$$

$$\frac{\Delta; \Gamma \vdash sv : \texttt{Vararg}_{c_1 \to c_2} \rightsquigarrow sv'}{\Delta; \Gamma \vdash \texttt{onearg}_{c_1 \to c_2}\, sv : c_1 \to c_2 \rightsquigarrow \texttt{call}\, \mathsf{onearg}[|c_1|, \mathrm{interp}\, |c_2|](\downharpoonleft c_1 \downharpoonright, sv')}$$

$$\frac{\Delta; \Gamma \vdash \mathit{fv} : \texttt{Float} \rightsquigarrow \mathit{fv}'}{\Delta; \Gamma \vdash \texttt{boxf}\, \mathit{fv} : \texttt{Boxf} \rightsquigarrow \texttt{box}\, \mathit{fv}'}$$

$$\frac{\Delta; \Gamma \vdash sv : \texttt{Sum}_i^j(\vec{c}) \rightsquigarrow sv'}{\Delta; \Gamma \vdash \texttt{proj}^j_{\texttt{Sum}_i(\vec{c})}\, sv : c_j \rightsquigarrow \texttt{select}^2\, sv'}$$

$$\frac{\Delta; \Gamma \vdash sv : c_j \rightsquigarrow sv'}{\Delta; \Gamma \vdash \texttt{inj}^j_{\texttt{Sum}_i(\vec{c})}\, sv : \texttt{Sum}_i^j(\vec{c}) \rightsquigarrow \texttt{inj\_union}_{|\texttt{Sum}_i(\vec{c})|}\langle \texttt{tag}_j, sv' \rangle}$$

$$\frac{\Delta;\Gamma \vdash sv_i : \tau_i \leadsto sv'_i}{\Delta;\Gamma \vdash \langle sv_1,\ldots,sv_n\rangle : \tau_1\times\ldots\times\tau_n \leadsto \langle sv'_1,\ldots,sv'_n\rangle}$$

$$\frac{\Delta;\Gamma \vdash sv : \tau_1\times\ldots\times\tau_n \leadsto sv'}{\Delta;\Gamma \vdash \texttt{select}^i\, sv : \tau_i \leadsto \texttt{select}^i\, sv'}$$

$$\frac{\Delta;\Gamma \vdash sv : \texttt{Sum}_i(\vec{c}) \leadsto sv' \quad \Delta;\Gamma[x_j : \texttt{Sum}_i^j(\vec{c})] \vdash e_j : \tau \leadsto e'_j}{\Delta;\Gamma \vdash \texttt{case}_\tau(sv)\,(x_1.e_1,\ldots,x_n.e_n) : \tau \leadsto \texttt{case}_{|\tau|}(sv')\,(x_1.e'_1,\ldots,x_n.e'_n)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \leadsto e'_1 \quad \Delta;\Gamma[x : \texttt{Exn}] \vdash e_2 : \tau \leadsto e'_2}{\Delta;\Gamma \vdash \texttt{handle}_\tau(e_1, x.e_2) : \tau \leadsto \texttt{handle}_{|\tau|}(e'_1, x.e'_2)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv : \forall[\alpha_1{::}\kappa_1,\ldots,\alpha_n{::}\kappa_n](\tau_1,\ldots,\tau_m)(k) \to \tau \leadsto sv' \\ \Delta;\Gamma \vdash sv_i : \tau_i[c_1/\alpha_1,\ldots,c_n/\alpha_n] \leadsto sv'_i \quad \Delta;\Gamma \vdash fv_i : \texttt{Float} \leadsto fv'_i\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash sv[c_1,\ldots,c_n](sv_1,\ldots,sv_m)(fv_1,\ldots,fv_k) : \tau[c_1/\alpha_1,\ldots,c_n/\alpha_n] \leadsto \\ \qquad sv'[|c_1|,\ldots,|c_n|](\lfloor c_1 \rfloor,\ldots,\lfloor c_n \rfloor, sv'_1,\ldots,sv'_m)(fv'_1,\ldots,fv'_k)\end{array}}$$

$$\frac{\Delta;\Gamma \vdash sv : \texttt{Exn} \leadsto sv'}{\Delta;\Gamma \vdash \texttt{raise}_\tau\, sv : \tau \leadsto \texttt{raise}_{|\tau|}\, sv'}$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \texttt{Dyntag}_c \leadsto sv'_1 \quad \Delta;\Gamma \vdash sv_2 : c \leadsto sv'_2}{\begin{array}{l}\Delta;\Gamma \vdash \texttt{inj\_dyn}_c(sv_1, sv_2) : \texttt{Exn} \leadsto \\ \quad \texttt{pack}\langle sv'_1, sv'_2\rangle\ \texttt{as}\ \exists\alpha{:}\mathrm{T}_{32}.(\texttt{Dyntag}(\alpha)\times\alpha)\ \texttt{hiding}\ |c|\end{array}}$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{mkexntag}_c : \texttt{Dyntag}_c \leadsto \texttt{dyntag}_{|c|}}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv_1 : \texttt{Dyntag}_{c_1} \leadsto sv'_1 \\ \Delta;\Gamma[x_1{:}c_1] \vdash e_1 : \tau \leadsto e'_1 \quad \Delta;\Gamma \vdash e_2 : \tau \leadsto e'_2\end{array}}{\Delta;\Gamma \vdash \texttt{exncase}_\tau(sv)\,(sv_1 \Rightarrow x_1.e_1, \_ \Rightarrow e_2) : \tau \leadsto \texttt{dyncase}_\tau(sv)\,(sv'_1 \Rightarrow x_1.e'_1, \_ \Rightarrow e'_2)}$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \texttt{Array}_c \leadsto sv'_1 \quad \Delta;\Gamma \vdash sv_2 : \texttt{Int} \leadsto sv'_2}{\Delta;\Gamma \vdash \texttt{sub}_c(sv_1, sv_2) : c \leadsto \texttt{call}\,\texttt{sub}[|c|](\lfloor c \rfloor, sv'_1, sv'_2)}$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \mathtt{Array}_c \rightsquigarrow sv_1' \quad \Delta;\Gamma \vdash sv_2 : \mathtt{Int} \rightsquigarrow sv_2' \quad \Delta;\Gamma \vdash sv_3 : c \rightsquigarrow sv_3'}{\Delta;\Gamma \vdash \mathtt{upd}_c(sv_1, sv_2, sv_3) : \mathtt{Unit} \rightsquigarrow \mathtt{call}\,\mathsf{upd}[|c|](\lfloor c\rfloor, sv_1', sv_2', sv_3')}$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \mathtt{Farray} \rightsquigarrow sv_1' \quad \Delta;\Gamma \vdash sv_2 : \mathtt{Int} \rightsquigarrow sv_2' \quad \Delta;\Gamma \vdash fv : \mathtt{Float} \rightsquigarrow fv'}{\Delta;\Gamma \vdash \mathtt{fupd}(sv_1, sv_2, fv) : \mathtt{Unit} \rightsquigarrow \mathtt{upd}_{\mathtt{Float}}(sv_1', sv_2', fv')}$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \mathtt{Int} \rightsquigarrow sv_1' \quad \Delta;\Gamma \vdash sv_2 : c \rightsquigarrow sv_2'}{\Delta;\Gamma \vdash \mathtt{array}_c(sv_1, sv_2) : \mathtt{Array}_c \rightsquigarrow \mathtt{call}\,\mathsf{array}[|c|](\lfloor c\rfloor, sv_1', sv_2')}$$

$$\frac{\Delta;\Gamma \vdash sv : \mathtt{Int} \rightsquigarrow sv' \quad \Delta;\Gamma \vdash fv : \mathtt{Float} \rightsquigarrow fv'}{\Delta;\Gamma \vdash \mathtt{farray}(sv, fv) : \mathtt{Farray} \rightsquigarrow \mathtt{array}_{\mathtt{Float}}(sv', fv')}$$

**Expressions** $\boxed{\Delta;\Gamma \vdash e : \tau \rightsquigarrow e'}$

$$\frac{\Delta;\Gamma \vdash sv : \tau \rightsquigarrow sv'}{\Delta;\Gamma \vdash sv : \tau \rightsquigarrow sv'}$$

$$\frac{\Delta;\Gamma \vdash i : \tau' \rightsquigarrow i' \quad \Delta;\Gamma[x : \tau'] \vdash e : \tau \rightsquigarrow e'}{\Delta;\Gamma \vdash \mathtt{let}_\tau\, x = i\, \mathtt{in}\, e : \tau \rightsquigarrow \mathtt{let}_{|\tau|}\, x = i'\, \mathtt{in}\, e'}$$

$$\frac{\Delta;\Gamma \vdash i : \mathtt{Float} \rightsquigarrow i' \quad \Delta;\Gamma[x_f] \vdash e : \tau \rightsquigarrow e'}{\Delta;\Gamma \vdash \mathtt{let}_\tau\, x_f = i\, \mathtt{in}\, e : \tau \rightsquigarrow \mathtt{let}_{|\tau|}\, x_f = i'\, \mathtt{in}\, e'}$$

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

# Chapter 6

# Closure converted LIL

## 6.1 Introduction

Closure conversion is a common strategy for implementing languages with higher order functions. The essential idea is to replace terms that evaluate via substitution of values for free variables with terms that can be evaluated by referring to an environment providing definitions for free variables. Since the latter evaluation model matches more closely that of actual machines, closure conversion is probably the most standard technique used for compiling higher-order functions.

In the **LIL**, closure conversion is the process by which function definitions are replaced with existentially abstracted pairs of environments and code pointers; and by which function applications are replaced with projections and code calls. Since the **LIL** is a typed language, the closure conversion translation must maintain well-typedness of terms. This means, among other things, that in addition to turning free *term* variables into additional function arguments, the closure conversion translation must also turn free *constructor* variables into additional function constructor arguments. However, since the **LIL** language admits a type erasure interpretation, the type environments do not need to be represented at runtime, which simplifies the problem of typed closure conversion.

The fact that **LIL** types can be erased at runtime means that in the underlying assembly language semantics, polymorphic instantiation can be a value, whereas in the **MIL** a polymorphic instantiation corresponds to passing actual runtime data and hence can not be thought of as a value in the sense of requiring no computation (as opposed to being *valuable* in the sense of being freely duplicable). This fact is important since it allows closure conversion to instantiate polymorphic functions directly when closures are built, which simplifies the theoretical framework for typed closure conversion.

There is extensive literature on typed closure conversion [CWM98, MWCG97, MMH96, MH98], and the **LIL** closure conversion algorithm adds nothing essential to this previous work. In this chapter I will briefly describe the important translation steps and prove a soundness theorem without going into any great detail.

## 6.2 The closure conversion translation

There are a number of different strategies possible for implementing closure conversion for recursive functions. Morrisett and Harper [MH98] give a fairly comprehensive overview of the different approaches, and in section 9.3 I discuss some of the trade-offs between the different approaches. The translation as described here uses the "recursive code" translation, which implements closures via simple existentials and pairs, but requires a primitive notion of self-calling code. Fortuitously (by design), the **LIL** language supports all of these notions.

In the next sections, I give a high-level overview of each part of the translation, and list a few key translation rules. In general, the closure conversion translation rewrites terms by first rewriting their sub-terms, and then reconstructing the term intact. Only in the specific cases of functions, function types, and function applications is a change made to the immediate structure of a term or type.

### 6.2.1 Constructors

Closure conversion does not change kinds in any way, so the first translation I discuss is the translation of constructors. I notate the translation of a **LIL** constructor $c$ under closure-conversion as $|c|$. Almost every case of the constructor translation is uninteresting except for the case for a polymorphic function type.

$$
\begin{array}{l}
|\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]. \rightarrow (c_1)(c_2)(c_3)| \stackrel{\text{def}}{=} \\
\quad \exists[\alpha_e{:}\mathrm{T}_{32}]. \times [\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]. \mathtt{Code}(\alpha_e :: |c_1|)(|c_2|)(|c_3|), \alpha_e]
\end{array}
$$

This case is actually quite illuminating since it clearly demonstrates the closure conversion process, and is therefore worth exploring more closely.

Recall the type of the arrow constructor in the **LIL**: $\rightarrow{:}\mathrm{T}_{32}\mathtt{list} \rightarrow \mathrm{T}_{64}\mathtt{list} \rightarrow \mathrm{T}_{32} \rightarrow \mathrm{T}_{32}$. The types $c_1$, $c_2$, and $c_3$ are therefore the the 32 bit argument type list, the 64 bit argument type list, and the return type, respectively. The translation of the arrow type simply translates these types compositionally, and then adds an additional type to the head of the 32 bit argument type list. This type, $\alpha_e$, is the type of the environment containing the free variables of the function that are to be passed to it as an additional argument. I use the informal :: notation to mean addition to the head of a list.

The type $\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]\,\mathtt{Code}(\alpha_e :: |c_1|)(|c_2|)(|c_3|)$ therefore describes a polymorphic code function which in addition to the 32 bit arguments described by $|c_1|$, expects a first argument described by $\alpha_e$. Referring to this type as $\tau_c$, the type $\times[\tau_c, \alpha_e]$ classifies a function of this type paired with an actual environment of the same type: $\alpha_e$.

What then is $\alpha_e$? The important point of the closure conversion translation in a typed setting is that the types of the free variables of a function are not apparent from its type. The environment type is *abstract* outside of the local scope where the function is defined. All that need be apparent from the type is that the type of the environment expected by the function and the type of the environment provided in the closure coincide.

In order to capture this abstraction, the standard technique is to use existential quantification. This then is the last part of the translation process. The final type is produced by existentially quantifying over the environment type, $\alpha_e$.

### 6.2.2 Typing contexts

The translation on typing contexts $\Psi$ and $\Gamma$ is defined in the obvious way by mapping the type translation across the ranges of the contexts.

### 6.2.3 Terms

| Term translations | | |
|---|---|---|
| Small values | $\Psi;\Delta;\Gamma \vdash sv : \tau \rightsquigarrow sv'$ | $\lvert\Psi\rvert;\Delta;\lvert\Gamma\rvert \vdash sv' : \lvert\tau\rvert$ |
| Float values | $\Psi;\Delta;\Gamma \vdash fv : \texttt{Float} \rightsquigarrow fv'$ | $\lvert\Psi\rvert;\Delta;\lvert\Gamma\rvert \vdash fv' : \texttt{Float}$ |
| Operations | $\Psi;\Delta;\Gamma \vdash opr : \tau \rightsquigarrow opr'$ | $\lvert\Psi\rvert;\Delta;\lvert\Gamma\rvert \vdash opr' : \lvert\tau\rvert\ \mathbf{opr}_{32}$ |
| 64 bit Operations | $\Psi;\Delta;\Gamma \vdash fopr : \texttt{Float} \rightsquigarrow fopr'$ | $\lvert\Psi\rvert;\Delta;\lvert\Gamma\rvert \vdash fopr' : \texttt{Float}\ \mathbf{opr}_{64}$ |
| Expression | $\Psi;\Delta;\Gamma \vdash e : \tau \rightsquigarrow d\,\texttt{in}\,e'$ | $\lvert\Psi\rvert,\Psi(d);\bullet;\bullet \vdash d\ \mathbf{ok}$<br>$\lvert\Psi\rvert,\Psi(d);\Delta;\lvert\Gamma\rvert \vdash e' : \lvert\tau\rvert\ \mathbf{exp}$ |
| Programs | $\Psi \vdash p : \tau \rightsquigarrow p'$ | $\Psi \vdash p' : \lvert\tau\rvert$ |

The high-level shape of the term level translation should be apparent from the discussion of the type translation. The essential task of the translation is to replace functions with existentially quantified code/environment pairs, and to replace function calls with closure-projections and code calls.

The term level translation is a typed translation, since the types of free variables is needed to construct the appropriate environment types. Additionally, it is very important in the **LIL** that the substitutions associated with type refinement operations be carried out as part of the translation, since these change the set of free type variables of expressions (as well as affecting equalities between them).

The general form of the closure conversion translation therefore exactly parallels the typing rules for the **LIL**. In addition to checking the type of terms however, the closure conversion relation also "produces" a new closure converted term. Moreover, in the case of the expression translation, a new set of heap values is also produced containing the new code replacing the functions from the original term.

The interesting cases from the translation arise from application and function abstraction. In order to simplify the syntactic structure of the translation, it is convenient to catch applications in the expression translation rather than in the more obvious operation translation. Consequently, the operation translation has no case for application.

#### Function application

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash g : \forall[\alpha_1{::}\kappa_1,\ldots,\alpha_n{::}\kappa_n](\tau_1,\ldots,\tau_m)(\phi_1,\ldots,\phi_n) \to \tau \rightsquigarrow sv' \\ \Psi;\Delta;\Gamma \vdash sv_i : \tau_i[c_1/\alpha_1,\ldots,c_n/\alpha_n] \rightsquigarrow sv'_i \quad \Psi;\Delta;\Gamma \vdash fv_i : \phi_i[c_1/\alpha_1,\ldots,c_n/\alpha_n] \rightsquigarrow fv'_i \\ \Psi;\Delta;\Gamma, x : \tau[c_1/\alpha_1,\ldots,c_n/\alpha_n] \vdash e : \tau' \rightsquigarrow d\,\texttt{in}\,e'\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma \vdash \texttt{let}\,x = g[c_1,\ldots,c_n](sv_1,\ldots,sv_m)(fv_1,\ldots,fv_k)\,\texttt{in}\,e : \tau' \rightsquigarrow \\ \quad d\,\texttt{in} \\ \qquad \texttt{let}[\alpha_e, x_c] = \texttt{unpack}\,sv'\,\texttt{in} \\ \qquad \texttt{let}\,f = \texttt{select}^1\,x_c\,\texttt{in} \\ \qquad \texttt{let}\,x_e = \texttt{select}^2\,x_c\,\texttt{in} \\ \qquad \texttt{let}\,x = \texttt{call}\,f[\lvert c_1\rvert,\ldots,\lvert c_n\rvert](x_e, sv'_1,\ldots,sv'_m)(fv'_1,\ldots,fv'_k)\,\texttt{in}\,e'\end{array}}$$

Since function definitions are replaced with closure definitions by the translation, the variable $g$ will be bound to a closure after the translation. The new term replacing the application first unpacks the closure to get at the underlying code/environment pair, and then selects out the components. The code pointer thus extracted is then passed the environment as an argument along with all of the original arguments.

#### Function abstraction

The translation of function definitions is by far the most complicated part of closure conversion. The complete translation rule is given in figure 6.1. Conceptually, it can be separated into two steps: producing code to construct the environment and the closure, and wrapping the function body in code to extract the components of the environment.

#### Creating the closure

The environment must contain all of the free variables of the function: that is, all of the free variables of the function body except the function arguments and the function itself. Since the **LIL** as defined lacks heterogeneous tuples, the 64 bit free variables must be boxed before they can be placed in the environment. Consequently, the closure creation code begins by boxing up all of the free 64 bit variables, then creates a record consisting of all the 32 bit variables followed by the boxed 64 bit variables. This record is the function's environment.

In addition to closing up the function over the free term variables via the environment, it is also necessary to abstract away all of the free type variables. These are "packaged" up into the closure by immediately instantiating the label for the new code function with the free type variables from the original function. Since the type parameters do not actually exist at runtime, there is no need to store them separately in the closure via a kind existential.

To create the final closure value, this partially instantiated label is written into a two field record with the environment which is then packed into the existential type, hiding the type of the environment. The original function name is bound to the resulting package.

#### Restoring free variables from the environment

The code from the previous section, executed at the site of the original function definition, serves to create the closure which is then passed around in place of the function. A second code sequence must also be added as a prelude to the function body to restore the free variables of the function from the environment. This code can be seen in the **fcode** definition from the function translation rule.

The initial segment of the un-packaging code simply projects out the 32 bit variables from the environment. The second segment projects out and un-boxes the boxed 64 bit variables from the environment. The final segment creates a closure to bind the function name to for internal uses. This consists of allocating a tuple to hold the code pointer and the environment, and packing it into the existential. The necessity of recreating the closure tuple inside of recursive functions is the principal disadvantage of the "recursive-code" approach to closure conversion. Note though that this is only necessary for *escaping* occurrences of the recursion variable: non-escaping occurrences can be replaced by uses of the code pointer and the original environment. This will be discussed in more detail in section 9.3.

### 6.2.4 Heap Values and Programs

Code functions are already closed, and hence are not directly closure-converted. However, in the **LIL** as defined, they may contain normal functions as sub-terms. Therefore, the bodies of code functions are translated just as any other expression.

$$
\frac{
\begin{array}{c}
\Delta \vdash \kappa_i\ \mathbf{ok} \quad (i \in 1 \ldots k) \\
\Delta, \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k \vdash \tau_i : \mathrm{T}_{32} \quad (i \in 1 \ldots m) \\
\Delta, \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k \vdash \phi_i : \mathrm{T}_{64} \quad (i \in 1 \ldots n) \\
\Delta, \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k \vdash \tau : \mathrm{T}_{32} \\
\Delta \vdash \tau_r \equiv \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k](\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n) \to \tau : \mathrm{T}_{32} \\
\Psi; \Delta, \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k; \Gamma, f{:}\tau, x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n \vdash e_f : \tau \leadsto d\, \mathtt{in}\, e'_f \\
\Psi; \Delta; \Gamma[f : \tau_r] \vdash e : \tau \leadsto d'\, \mathtt{in}\, e' \\
fv_{32}(e_f) \setminus \{x_1, \ldots, x_m, f\} = \{y_1, \ldots, y_q\} \qquad fv_{64}(e_f) \setminus \{z_1, \ldots, z_n\} = \{r_1, \ldots, r_p\} \\
fv_t(e_f) \setminus \{\alpha_1, \ldots, \alpha_k\} = \{\beta_1, \ldots, \beta_n\} \\
\tau_e \stackrel{\text{def}}{=} \times[|\Gamma(y_1)|, \ldots, |\Gamma(y_q)|, \mathtt{Boxed}\,|\Gamma(r_1)|, \ldots, \mathtt{Boxed}\,|\Gamma(r_p)|] \\
\begin{array}{rl}
\mathbf{fcode} \stackrel{\text{def}}{=} & \mathtt{code}[\beta_1{:}\Delta(\beta_1), \ldots, \beta_l{:}\Delta(\beta_l), \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k] \\
& (x_e{:}\tau_e, x_1{:}|\tau_1|, \ldots, x_m{:}|\tau_m|) \\
& (z_1{:}|\phi_1|, \ldots, z_n{:}|\phi_n|). \\
& \quad \mathtt{let}\, y_1 = \mathtt{select}^1\, x_e\, \mathtt{in} \\
& \quad \cdots \\
& \quad \mathtt{let}\, y_q = \mathtt{select}^q\, x_e\, \mathtt{in} \\
& \quad \mathtt{let}\, b_1 = \mathtt{select}^{q+1}\, x_e\, \mathtt{in} \\
& \quad \mathtt{let}\, r_1 = \mathtt{unbox}\, b_1\, \mathtt{in} \\
& \quad \cdots \\
& \quad \mathtt{let}\, b_p = \mathtt{select}^{q+p+1}\, x_e\, \mathtt{in} \\
& \quad \mathtt{let}\, r_p = \mathtt{unbox}\, b_p\, \mathtt{in} \\
& \quad \mathtt{let}\, f_c = \langle \mathsf{fcode}[\beta_1, \ldots, \beta_l], x_e \rangle\, \mathtt{in} \\
& \quad \mathtt{let}\, f = \mathtt{pack}\, f_c\, \mathtt{as}\, |\tau|\, \mathtt{hiding}\, \tau_e\, \mathtt{in} \\
& \quad e'_f
\end{array}
\end{array}
}{
\begin{array}{l}
\Psi; \Delta; \Gamma \vdash \mathtt{let}_\tau\, \mathtt{rec}_{\tau_r}\, f[\vec{\alpha{:}\kappa}](\vec{x{:}\tau})(\vec{x_f}).e_f\, \mathtt{in}\, e : \forall[\vec{\alpha{:}\kappa}](\vec{\tau})(|x_f|) \to \tau \leadsto \\
\qquad d, \mathsf{fcode}{:}|\tau_r| \mapsto \mathbf{fcode}, d' \\
\qquad\quad \mathtt{let}\, b_1 = \mathtt{box}\, r_1\, \mathtt{in} \\
\qquad\quad \cdots \\
\qquad\quad \mathtt{let}\, b_p = \mathtt{box}\, r_p\, \mathtt{in} \\
\qquad\quad \mathtt{let}\, x_e = \langle y_1, \ldots, y_q, b_1, \ldots, b_p \rangle\, \mathtt{in} \\
\qquad\quad \mathtt{let}\, f_c = \langle \mathsf{fcode}[\beta_1, \ldots, \beta_l], x_e \rangle\, \mathtt{in} \\
\qquad\quad \mathtt{let}\, f = \mathtt{pack}\, f_c\, \mathtt{as}\, |\tau|\, \mathtt{hiding}\, \tau_e\, \mathtt{in} \\
\qquad\quad e'
\end{array}
}
$$

**Figure 6.1:** Closure converting recursive function definitions

$$\frac{\Psi; \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k; x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n \vdash e : \tau \leadsto d\,\mathtt{in}\,e'}{\begin{array}{l}\Psi \vdash \mathtt{code}[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k](x_1{:}\tau_1, \ldots, x_m{:}\tau_m)(z_1{:}\phi_1, \ldots, z_n{:}\phi_n).e : \\ \qquad\qquad\qquad \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k]\,\mathtt{Code}(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n)(\tau) \leadsto \\ \quad d\,\mathtt{in}\,\mathtt{code}_{|\tau|}[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k](x_1{:}|\tau_1|, \ldots, x_m{:}|\tau_m|)(z_1{:}|\phi_1|, \ldots, z_n{:}|\phi_n|).e'\end{array}}$$

Heaps are translated by translating the individual heap values, lifting out any new heap values to the top level.

$$\frac{}{\Psi \vdash \epsilon \leadsto \epsilon\,\mathtt{in}\,\epsilon}$$

$$\frac{\Psi[\ell{:}\tau] \vdash hval : \tau \leadsto d_\ell\,\mathtt{in}\,hval' \quad \Psi[\ell{:}\tau] \vdash d \leadsto d_h\,\mathtt{in}\,d'}{\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval \leadsto d_\ell, d_h\,\mathtt{in}\,d', \ell{:}|\tau| \mapsto hval'}$$

The translation on programs rewrites the heap and the expression body to form a new program.

$$\frac{\begin{array}{c}\Psi, \Psi(d) \vdash d \leadsto d_h\,\mathtt{in}\,d' \\ \Psi, \Psi(d); \bullet; \bullet \vdash e : \tau \leadsto d_e\,\mathtt{in}\,e'\end{array}}{\Psi \vdash \mathtt{letrec}\,d\,\mathtt{in}\,e : \tau \leadsto \mathtt{letrec}\,d', d_h, d_e\,\mathtt{in}\,e'}$$

## 6.3 Soundness of closure conversion

The proof of type preservation for the closure conversion translation is fairly straightforward, as only a few sub-terms are actually changed. In general, most of the cases follow directly by induction, reconstructing the original term or derivation from the inductively re-written sub-terms or sub-derivations. Only in the few cases where the immediate term is re-written does the proof involve anything more substantial.

### 6.3.1 Constructors

As with the **MIL** to **LIL** translation, I begin by proving the soundness of the constructor translation, and then proceed with auxiliary lemmas showing that the constructor translation commutes with substitution, and respects equivalence. The latter two lemmas are useful for the term level proofs.

**Theorem 10 (Soundness of the constructor translation)**
*If* $\Delta \vdash c : \kappa$ *then* $\Delta \vdash |c| : \kappa$*.*

**Proof:** By induction on the structure of types. For all types and constructors except the type of polymorphic functions, the translation leaves the structure of the type the same, and so the proof follows straightforwardly by induction, producing the new derivation from the inductively obtained sub-derivations using the same rule as in the original derivation. The only interesting rule is the rule for the type of polymorphic functions.

Suppose the type being translated is of the form $\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\rightarrow(c_1)(c_2)(c_3)$.
By inversion of the assumed derivation:

$\Delta \vdash \kappa_i$ **ok** $\quad i \in 1\ldots n$
$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n \vdash c_1 : \mathrm{T}_{32}\texttt{list}$
$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n \vdash c_2 : \mathrm{T}_{32}\texttt{list}$
$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n \vdash c_3 : \mathrm{T}_{32}$

By induction:

$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n \vdash |c_1| : \mathrm{T}_{32}\texttt{list}$
$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n \vdash |c_2| : \mathrm{T}_{32}\texttt{list}$
$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n \vdash |c_3| : \mathrm{T}_{32}$

By weakening:

$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n,\alpha_e{:}\mathrm{T}_{32} \vdash |c_1| : \mathrm{T}_{32}\texttt{list}$
$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n,\alpha_e{:}\mathrm{T}_{32} \vdash |c_2| : \mathrm{T}_{32}\texttt{list}$
$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n,\alpha_e{:}\mathrm{T}_{32} \vdash |c_3| : \mathrm{T}_{32}$
(Note that $\alpha_e$ can always be chosen appropriately)

By construction (cons):

$\Delta,\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n,\alpha_e{:}\mathrm{T}_{32} \vdash (\alpha_e :: |c_1|) : \mathrm{T}_{32}\texttt{list}$

By construction (application and universal introduction):

$\Delta,\alpha_e{:}\mathrm{T}_{32} \vdash \forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\,\texttt{Code}(\alpha_e :: |c_1|)(|c_2|)(|c_3|) : \mathrm{T}_{32}\texttt{list}$

By construction (pairing and existential introduction):

$\Delta \vdash \exists[\alpha_e{:}\mathrm{T}_{32}].\times[\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\,\texttt{Code}(\alpha_e :: |c_1|)(|c_2|)(|c_3|),\alpha_e] : \mathrm{T}_{32}$

■

**Lemma 23 (The type translation commutes with substitution)**
$|c[c'/\alpha]| = |c|[|c'|/\alpha]$

**Proof:** By induction on $c$.

1. If $c$ is a constant, then $\alpha \notin \mathit{fv}c, \mathit{fv}|c|$, so $|c|[|c'|/\alpha] = |c| = |c[c'/\alpha]|$.
2. If $c$ is a variable $\alpha'$:
   (a) If $\alpha \neq \alpha'$ then as in the previous case.
   (b) If $\alpha = \alpha'$ then $|\alpha|[|c'|/\alpha] = \alpha[|c'|/\alpha] = |c'| = |\alpha[c'/\alpha]|$.
3. If $c = \lambda(\beta{:}\kappa).c_2$, then by definition

   $$|\lambda(\beta{:}\kappa).c_2|[|c'|/\alpha] = (\lambda(\beta{:}|\kappa|).|c_2|)[|c'|/\alpha] = \lambda(\beta{:}|\kappa|).(|c_2|[|c'|/\alpha])$$

   By induction and the definition of substitution:

   $$\lambda(\beta{:}|\kappa|).(|c_2|[|c'|/\alpha]) = \lambda(\beta{:}|\kappa|).(|c_2[c'/\alpha]|) = |\lambda(\beta{:}\kappa).(c_2[c'/\alpha])|$$

   Note, I rely on alpha-variance to ensure non-capture.
4. If $c = \forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\rightarrow(c_1)(c_2)(c_3)$ then by the definition of the translation and of substitution

   $|\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\rightarrow(c_1)(c_2)(c_3)|[|c'|/\alpha]$
   $\stackrel{\text{def}}{=} (\exists[\alpha_e{:}\mathrm{T}_{32}].\times[\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\,\texttt{Code}(\alpha_e :: |c_1|)(|c_2|)(|c_3|),\alpha_e])[|c'|/\alpha]$
   $\stackrel{\text{def}}{=} \exists[\alpha_e{:}\mathrm{T}_{32}].\times[\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\,\texttt{Code}(\alpha_e :: |c_1|[|c'|/\alpha])(|c_2|[|c'|/\alpha])(|c_3|[|c'|/\alpha]),\alpha_e]$

Note that I rely on alpha-variance to avoid capture.

Finally, by induction, and by the definition of substitution:

$$\begin{aligned}
&\exists[\alpha_e{:}\mathrm{T}_{32}].\times[\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\,\mathtt{Code}(\alpha_e :: |c_1|[|c'|/\alpha])(|c_2|[|c'|/\alpha])(|c_3|[|c'|/\alpha]),\alpha_e]\\
&\stackrel{\text{def}}{=} \exists[\alpha_e{:}\mathrm{T}_{32}].\times[\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\,\mathtt{Code}(\alpha_e :: |c_1[c'/\alpha]|)(|c_2[c'/\alpha]|)(|c_3[c'/\alpha]|),\alpha_e]\\
&\stackrel{\text{def}}{=} |\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_n{:}\kappa_n].\rightarrow(c_1)(c_2)(c_3)[c'/\alpha]|
\end{aligned}$$

5. If $c = c_1c_2, \pi_i c, \langle c_1, c_2\rangle$, the proof proceeds similarly.

■

**Lemma 24 (The type translation respects equivalence)**
*If* $\Delta \vdash c_1 \equiv c_2 : \kappa$ *then* $\Delta \vdash |c_1| \equiv |c_2| : \kappa$.

**Proof:** (By induction on equivalence derivations) By induction on equivalence derivations. The proof is straightforward with almost all cases identical to the proof of lemma 10 from chapter 5. The only cases where the derivation changes is in the case that the last rule used is the structural rule for constructor application $c_1\ c_2$. Inductively, equivalence derivations exist for the sub-terms. If the applications match the form of a polymorphic function (that is, $\forall[\kappa_1](\ldots(\forall[\kappa_n](\rightarrow(c_x)(c_z)(c_r)))))$, then the application, reflexivity, pair, and existential equivalence rules must be applied to construct the equivalence derivation for the translation. If the applications do not match such a form, then the application rule suffices. All of the primitive types follow directly by reflexivity or by the structural application rule. I give some example cases here.

1. Suppose $\Delta \vdash c \equiv c : \kappa$ by reflexivity.

   By assumption:
   $\Delta \vdash c : \kappa$

   By theorem 10:
   $\Delta \vdash |c| : \kappa$

   By reflexivity:
   $\Delta \vdash |c| \equiv |c| : \kappa$

2. Suppose $\Delta \vdash (\lambda(\alpha{:}\kappa_1).c_1)(c_2) \equiv c_1[c_2/\alpha] : \kappa_2$.

   By assumption:
   $\Delta \vdash \kappa_1$ **ok**
   $\Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa_2$
   $\Delta \vdash c_2 : \kappa_1$

   By assumption, and by theorem 2:
   $\Delta \vdash \kappa_1$ **ok**
   $\Delta, \alpha{:}\kappa_1 \vdash |c_1| : \kappa_2$
   $\Delta \vdash |c_2| : \kappa_1$

   By the $\lambda$ beta rule and the definition of the translations:
   $\Delta \vdash |(\lambda(\alpha{:}\kappa_1).c_1)(c_2)| \equiv |c_1|[|c_2|/\alpha] : \kappa_2$

   Finally, by lemma 9:
   $\Delta \vdash |(\lambda(\alpha{:}\kappa_1).c_1)(c_2)| \equiv |c_1[c_2/\alpha]| : \kappa_2$

3. Suppose $\Delta \vdash \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\rightarrow(c_1)(c_2)(c_3) \equiv \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\rightarrow(c_1')(c_2')(c_3') : \mathrm{T}_{32}$.

   By induction and weakening
   $\Delta, \alpha_e{:}\mathrm{T}_{32}, \alpha_1{:}\kappa_1 \ldots \alpha_n{:}\kappa_n \vdash c_1 \equiv c_1' : \mathrm{T}_{32}\mathtt{list}$
   $\Delta, \alpha_e{:}\mathrm{T}_{32}, \alpha_1{:}\kappa_1 \ldots \alpha_n{:}\kappa_n \vdash c_2 \equiv c_2' : \mathrm{T}_{32}\mathtt{list}$
   $\Delta, \alpha_e{:}\mathrm{T}_{32}, \alpha_1{:}\kappa_1 \ldots \alpha_n{:}\kappa_n \vdash c_3 \equiv c_3' : \mathrm{T}_{32}$

   By reflexivity
   $\Delta, \alpha_e{:}\mathrm{T}_{32}, \alpha_1{:}\kappa_1 \ldots \alpha_n{:}\kappa_n \vdash \alpha_e \equiv \alpha_e : \mathrm{T}_{32}$

   By the pair and existential rules
   $\Delta \vdash \exists[\alpha_e{:}\mathrm{T}_{32}].\times[\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\mathtt{Code}(\alpha_e :: |c_1|)(|c_2|)(|c_3|), \alpha_e]$
   $\equiv \exists[\alpha_e{:}\mathrm{T}_{32}].\times[\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\mathtt{Code}(\alpha_e :: |c_1'|)(|c_2'|)(|c_3'|), \alpha_e] : \mathrm{T}_{32}$

■

### 6.3.2 Typing contexts

**Theorem 11 (Soundness of the context translations)**

1. *If* $\Delta \vdash \Gamma$ **ok** *then* $\Delta \vdash |\Gamma|$ **ok**.

2. *If* $\vdash \Psi$ **ok** *then* $\vdash |\Psi|$ **ok**.

**Proof:** By induction on typing contexts. The proof proceeds by cases on contexts.

1. Suppose $\Delta \vdash \Gamma$ **ok**.

   - If $\Gamma = \bullet$, then its translation is empty, and hence is trivially well-formed.
   - If $\Gamma = \Gamma', x{:}\tau$, then by induction, $\Delta \vdash |\Gamma'|$ **ok**. By theorem 10, $\Delta \vdash |\tau|$ **ok**, and since the domain is unchanged by the translation, $x \notin \Gamma'$. Therefore, by construction $\Delta \vdash |\Gamma', x{:}|\tau||$ **ok**.

2. The proof for heap contexts $\Psi$ proceeds identically.

■

### 6.3.3 Terms

For terms, I begin by stating the soundness theorem for values and 64 bit operations. For these syntactic classes, the statement of the theorem is straightforward.

**Theorem 12 (Values and 64 bit operations)**

1. *If* $\Psi; \Delta; \Gamma \vdash sv : \tau$ *and* $\Psi; \Delta; \Gamma \vdash sv : \tau \rightsquigarrow sv'$ *then* $|\Psi|; \Delta; |\Gamma| \vdash sv' : |\tau|$.

2. *If* $\Psi; \Delta; \Gamma \vdash fv : \mathtt{Float}$ *and* $\Psi; \Delta; \Gamma \vdash fv : \mathtt{Float} \rightsquigarrow fv'$ *then* $|\Psi|; \Delta; |\Gamma| \vdash fv' : \mathtt{Float}$.

3. *If* $\Psi; \Delta; \Gamma \vdash fopr : \mathtt{Float}\ \mathbf{opr}_{64}$ *and* $\Psi; \Delta; \Gamma \vdash fopr : \mathtt{Float} \rightsquigarrow fopr'$ *then*

$$|\Psi|; \Delta; |\Gamma| \vdash fopr' : \mathtt{Float}\ \mathbf{opr}_{64}$$

**Proof (Sketch):** By induction on terms. The closure conversion translation for each of these syntactic classes leaves the top level structure of the term intact, reconstructing it from the inductively re-written sub-terms. Therefore, the proof constructs a new derivation in each case by using the same inference rule from the original derivation on the inductively re-written sub-terms. Kinds are unchanged by the translation, and for constructors I appeal to theorem 10.

■

The theorem for 32 bit operations and expressions is more complicated, since the expression translation potentially yields new heap bindings (code functions) to be lifted out. The soundness theorem for expressions states that both the bindings and the new expression are well-formed in the heap context obtained by extending the translation of the original heap context with the bindings for the additional heap values produced by the translation.

**Theorem 13 (32 bit operations and expressions)**

1. *If*

$$\Psi; \Delta; \Gamma \vdash opr : \tau\ \mathbf{opr}_{32} \quad \textit{and} \quad \Psi; \Delta; \Gamma \vdash opr : \tau \leadsto opr'$$

*then*

$$|\Psi|; \Delta; |\Gamma| \vdash opr' : |\tau|\ \mathbf{opr}_{32}$$

2. *If*

$$\Psi; \Delta; \Gamma \vdash e : \tau\ \mathbf{exp} \quad \textit{and} \quad \Psi; \Delta; \Gamma \vdash e : \tau \leadsto d\,\mathtt{in}\,e'$$

*then*

$$|\Psi|, \Psi(d) \vdash d\ \mathbf{ok} \quad \textit{and} \quad |\Psi|, \Psi(d); \Delta; |\Gamma| \vdash \mathtt{in}\,e' : |\tau|\ \mathbf{exp}$$

**Proof:** By induction on terms.

1. The proof for operations proceeds exactly as in the previous theorem, appealing to the induction hypothesis to obtain well-typedness derivations for sub-terms, and constructing a new derivation with the same rule as in the original derivation. Note that the case for applications is trivially true, since the operation translation is not defined on applications.

2. The proof for expressions also proceeds similarly, except in the cases binding applications and functions where the translation does incremental work.

   Suppose $\mathtt{let}\,x = g[c_1, \ldots, c_n](sv_1, \ldots, sv_m)(fv_1, \ldots, fv_k)\,\mathtt{in}\,e$

   By assumption:
   $\Psi; \Delta; \Gamma \vdash sv : \forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_k) \to \tau$
   $\Delta \vdash c_i : \kappa_i \quad i \in 1 \ldots n$
   $\Psi; \Delta; \Gamma \vdash sv_i : \tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n] \quad i \in 1 \ldots m$
   $\Psi; \Delta; \Gamma \vdash fv_i : \phi_i[c_1/\alpha_1, \ldots, c_n/\alpha_n] \quad i \in 1 \ldots k$
   $\Psi; \Delta; \Gamma, x{:}\tau[c_1/\alpha_1, \ldots, c_n/\alpha_n] \vdash e : \tau'\ \mathbf{exp}$
   $\Psi; \Delta; \Gamma \vdash g : \forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau \leadsto sv'$
   $\Psi; \Delta; \Gamma \vdash sv_i : \tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n] \leadsto sv'_i$
   $\Psi; \Delta; \Gamma \vdash fv_i : \phi_i[c_1/\alpha_1, \ldots, c_n/\alpha_n] \leadsto fv'_i$
   $\Psi; \Delta; \Gamma, x : \tau[c_1/\alpha_1, \ldots, c_n/\alpha_n] \vdash e : \tau' \leadsto d\,\mathtt{in}\,e'$

It suffices to show that:

$$
\begin{array}{lll}
|\Psi|, \Psi(d); \Delta; |\Gamma| \vdash & \texttt{let}[\alpha_e, x_c] = \texttt{unpack}\, sv' \,\texttt{in} & : |\tau'| \;\mathbf{exp} \\
& \texttt{let}\, f = \texttt{select}^1\, x_c \,\texttt{in} & \\
& \texttt{let}\, x_e = \texttt{select}^2\, x_c \,\texttt{in} & \\
& \texttt{let}\, x = \texttt{call}\, f[|c_1|, \ldots, |c_n|](x_e, sv'_1, \ldots, sv'_m)(fv'_1, \ldots, fv'_k) & \\
& \texttt{in}\, e' &
\end{array}
$$

Let

$$\tau_c = \times[\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\, \texttt{Code}(\alpha_e, |\tau_1|, \ldots, |\tau_m|)(|\phi_1|, \ldots, |\phi_k|)(|\tau|), \alpha_e]$$

and

$$\tau_f = \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\, \texttt{Code}(\alpha_e, |\tau_1|, \ldots, |\tau_m|)(|\phi_1|, \ldots, |\phi_k|)(|\tau|)$$

.

So by the expression typing rules, it suffices to show:

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma| \vdash sv' : \exists[\alpha_e{:}\mathrm{T}_{32}].\tau_c$

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c \vdash \texttt{select}^1\, x_c : \tau_f \;\mathbf{opr}_{32}$

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c \vdash \texttt{select}^2\, x_c : \alpha_e \;\mathbf{opr}_{32}$

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c, f{:}\tau_f, x_e{:}\alpha_e \vdash \texttt{call}\, f[|\vec{c_i}|](x_e, \vec{sv'_j})(\vec{fv'_l}) : |\tau|[\overrightarrow{|c_i|/\alpha_i}] \;\mathbf{opr}_{32}$

$\quad i \in 1 \ldots n, j \in 1 \ldots m, l \in 1 \ldots k$

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c, f{:}\tau_f, x_e{:}\alpha_e, x{:}|\tau|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] \vdash e' : |\tau'| \;\mathbf{exp}$

By theorem 12:

$|\Psi|, \Psi(d); \Delta; |\Gamma| \vdash sv : |\forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_k) \to \tau|$

And by definition:

$|\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_k) \to \tau| \stackrel{\text{def}}{=}$

$\quad \exists[\alpha_e{:}\mathrm{T}_{32}]. \times [\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\, \texttt{Code}(\alpha_e, |\tau_1|, \ldots, |\tau_m|)(|\phi_1|, \ldots, |\phi_k|)(|\tau|), \alpha_e]$

So by the unpack rule:

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma| \vdash sv' : \exists[\alpha_e{:}\mathrm{T}_{32}].\tau_c \quad$ For some $\tau_c$

By the variable rule:

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c \vdash x_c : \times[\tau_f, \alpha_e]$

So by the select rule:

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c \vdash \texttt{select}^1\, x_c : \tau_f \;\mathbf{opr}_{32}$

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c \vdash \texttt{select}^2\, x_c : \alpha_e \;\mathbf{opr}_{32}$

By the variable rule:

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c, f{:}\tau_f, x_e{:}\alpha_e \vdash f : \tau_f$

By theorems 10 and 12:

$\Delta \vdash |c_i| : \kappa_i \quad i \in 1 \ldots n$

$|\Psi|, \Psi(d); \Delta; |\Gamma| \vdash sv'_i : |\tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n]| \quad i \in 1 \ldots m$

$|\Psi|, \Psi(d); \Delta; |\Gamma| \vdash fv'_i : |\phi_i[c_1/\alpha_1, \ldots, c_n/\alpha_n]| \quad i \in 1 \ldots k$

By the commutation lemma (lemma 23):

$|\Psi|, \Psi(d); \Delta; |\Gamma| \vdash sv'_i : |\tau_i|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] \quad i \in 1 \ldots m$

$|\Psi|, \Psi(d); \Delta; |\Gamma| \vdash fv'_i : |\phi_i|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] \quad i \in 1 \ldots k$

Therefore by the `call` rule :

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c, f{:}\tau_f, x_e{:}\alpha_e \vdash \texttt{call}\, f[|\vec{c_i}|](x_e, \vec{sv'_j})(\vec{fv'_l}) : |\tau|[\overrightarrow{|c_i|/\alpha_i}]\ \mathbf{opr}_{32}$
$i \in 1 \ldots n, j \in 1 \ldots m, l \in 1 \ldots k$

Finally, by induction:

$|\Psi|, \Psi(d); \Delta, \alpha_e{:}\mathrm{T}_{32}; |\Gamma|, x_c{:}\tau_c, f{:}\tau_f, x_e{:}\alpha_e, x{:}|\tau|[|c_1|/\alpha_1, \ldots, |c_n|/\alpha_n] \vdash e' : |\tau'|\ \mathbf{exp}$

3. The case for function abstraction proceeds in a similar fashion, building up well-typedness derivations for the **fcode** definitions and for the closure creation code.

■

An additional lemma relating translations of heap contexts to translations of heaps is needed for subsequent theorems.

**Lemma 25 ($\Psi(d)$)**
*If* $\Psi \vdash d \rightsquigarrow d_h\, \texttt{in}\, d'$ *then* $|\Psi(d)| = \Psi(d')$.

**Proof:** By induction on d.

1. If $d = \epsilon$ then $d' = \epsilon$ and $|\Psi(d)| = \epsilon = \Psi(d')$.
2. If $d = d_r, \ell{:}\tau \mapsto hval$ then:

By assumption:

$\Psi[\ell{:}\tau] \vdash hval : \tau \rightsquigarrow d_\ell\, \texttt{in}\, hval'$
$\Psi[\ell{:}\tau] \vdash d_r \rightsquigarrow d_h\, \texttt{in}\, d'_r$
$\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval \rightsquigarrow d_\ell, d_h\, \texttt{in}\, d'_r, \ell{:}|\tau| \mapsto hval'$

By definition:

$|\Psi(d)|$
$= |\Psi(d_r, \ell{:}\tau \mapsto hval)|$
$= |\Psi(d_r), \ell{:}\tau|$
$= |\Psi(d_r)|, \ell{:}|\tau|$

By induction:

$|\Psi(d_r)| = \Psi(d'_r)$
so
$|\Psi(d_r)|, \ell{:}|\tau|$
$= \Psi(d'_r), \ell{:}|\tau|$
$= \Psi(d'_r, \ell{:}|\tau| \mapsto hval')$

■

As with expressions, translating heap values may yield additional heap bindings. The soundness theorem for the heap value translation states that the both the new heap value and the new bindings are well-formed in the translation of the original context extended with bindings for the new heap entries.

**Lemma 26 (Soundness of the heap value translation)**
*If*

$\Psi \vdash hval : \tau\ \mathbf{hval}$ *and* $\Psi \vdash hval : \tau \rightsquigarrow d\, \texttt{in}\, hval'$

*then*

$|\Psi|, \Psi(d) \vdash d\ \mathbf{ok}$ *and* $|\Psi|, \Psi(d) \vdash hval' : |\tau|\ \mathbf{hval}$

**Proof:** By construction.
By assumption:

$\Psi \vdash \mathtt{code}_\tau[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k](x_1{:}\tau_1,\ldots,x_m{:}\tau_m)(z_1{:}\phi_1,\ldots,z_n{:}\phi_n).e :$
$\quad \forall[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k]\,\mathtt{Code}(\tau_1,\ldots,\tau_m)(\phi_1,\ldots,\phi_n)(\tau)\ \textbf{hval}$

and

$\Psi \vdash \mathtt{code}_\tau[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k](x_1{:}\tau_1,\ldots,x_m{:}\tau_m)(z_1{:}\phi_1,\ldots,z_n{:}\phi_n).e :$
$\quad \forall[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k]\,\mathtt{Code}(\tau_1,\ldots,\tau_m)(\phi_1,\ldots,\phi_n)(\tau) \rightsquigarrow$
$\quad d\,\mathtt{in}\,\mathtt{code}_{|\tau|}[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k](x_1{:}|\tau_1|,\ldots,x_m{:}|\tau_m|)(z_1{:}|\phi_1|,\ldots,z_n{:}|\phi_n|).e'$

By inversion:

$\bullet \vdash \kappa_i\ \textbf{ok} \quad (i \in 1\ldots k)$
$\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k \vdash \tau_i : \mathrm{T}_{32} \quad (i \in 1\ldots m)$
$\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k \vdash \phi_i : \mathrm{T}_{64} \quad (i \in 1\ldots n)$
$\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k \vdash \tau : \mathrm{T}_{32}$
$\Psi; \alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k; x_1{:}\tau_1,\ldots,x_m{:}\tau_m, z_1{:}\phi_1,\ldots,z_n{:}\phi_n \vdash e : \tau\ \textbf{exp}$
$\Psi; \alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k; x_1{:}\tau_1,\ldots,x_m{:}\tau_m, z_1{:}\phi_1,\ldots,z_n{:}\phi_n \vdash e : \tau \rightsquigarrow d\,\mathtt{in}\,e'$

By theorem 10:

$\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k \vdash |\tau_i| : \mathrm{T}_{32} \quad (i \in 1\ldots m)$
$\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k \vdash |\phi_i| : \mathrm{T}_{64} \quad (i \in 1\ldots n)$
$\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k \vdash |\tau| : \mathrm{T}_{32}$

By theorem 13:

$|\Psi|, \Psi(d) \vdash d\ \textbf{ok}$ and
$|\Psi|, \Psi(d); \alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k; x_1{:}|\tau_1|,\ldots,x_m{:}|\tau_m|, z_1{:}|\phi_1|,\ldots,z_n{:}|\phi_n| \vdash e' : |\tau|\ \textbf{exp}$

Note that:

$|\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k]\,\mathtt{Code}(\tau_1,\ldots,\tau_m)(\phi_1,\ldots,\phi_n)(\tau)|$
$= \forall[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k]\,\mathtt{Code}(|\tau_1|,\ldots,|\tau_m|)(|\phi_1|,\ldots,|\phi_n|)(|\tau|)$

So by the heap value rule :

$|\Psi|, \Psi(d) \vdash \mathtt{code}_{|\tau|}[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k](x_1{:}|\tau_1|,\ldots,x_m{:}|\tau_m|)(z_1{:}|\phi_1|,\ldots,z_n{:}|\phi_n|).e' :$
$\quad |\forall[\alpha_1{:}\kappa_1,\ldots,\alpha_k{:}\kappa_k]\,\mathtt{Code}(\tau_1,\ldots,\tau_m)(\phi_1,\ldots,\phi_n)(\tau)|\ \textbf{hval}$

∎

**Lemma 27 (Soundness of the heap translation)**
*If* $\Psi \vdash d\ \textbf{ok}$ *and* $\Psi \vdash d \rightsquigarrow d_h\,\mathtt{in}\,d'$ *then* $|\Psi|, \Psi(d_h) \vdash d_h, d'\ \textbf{ok}$.

**Proof:** By induction on $d$. We proceed by cases on $d$.

1. $\epsilon$

   By assumption:
   $\vdash \Psi\ \textbf{ok}$

   By theorem 11:
   $\vdash |\Psi|\ \textbf{ok}$

   By the empty heap rule:
   $|\Psi| \vdash \epsilon\ \textbf{ok}$
   (Note that $\Psi(\epsilon) = \bullet$)

2. $d, \ell{:}\tau \mapsto \mathit{hval}$

By assumption:
$\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval$ **ok**
$\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval \rightsquigarrow d_\ell, d_h\, \mathtt{in}\, d', \ell{:}|\tau| \mapsto hval'$

By inverting assumptions:
$\Psi[\ell{:}\tau] \vdash hval : \tau$ **hval**
$\Psi[\ell{:}\tau] \vdash hval : \tau \rightsquigarrow d_\ell\, \mathtt{in}\, hval'$
$\Psi[\ell{:}\tau] \vdash d$ **ok**
$\Psi[\ell{:}\tau] \vdash d \rightsquigarrow d_h\, \mathtt{in}\, d'$

By induction:
$|\Psi[\ell{:}\tau]|, \Psi(d_h) \vdash d_h, d'$ **ok**

By lemma 26:
$|\Psi[\ell{:}\tau]|, \Psi(d_\ell) \vdash d_\ell$ **ok**and
$|\Psi[\ell{:}\tau]|, \Psi(d_\ell) \vdash hval' : |\tau|$ **hval**

By lemma weakening:
$|\Psi[\ell{:}\tau]|, \Psi(d_h), \Psi(d_\ell) \vdash d_h, d'$ **ok**
$|\Psi[\ell{:}\tau]|, \Psi(d_h), \Psi(d_\ell) \vdash d_\ell$ **ok**

By definition:
$|\Psi[\ell{:}\tau]|, \Psi(d_h), \Psi(d_\ell) \vdash d_h, d', d_\ell$ **ok**
and$|\Psi[\ell{:}\tau]| = |\Psi|[\ell{:}|\tau|]$

So by the non-empty heap rule:
$|\Psi[\ell{:}\tau]|, \Psi(d_\ell, d_h) \vdash d_\ell, d_h, d', \ell{:}|\tau| \mapsto hval'$ **ok**

■

**Theorem 14 (Programs)**
*If* $\Psi \vdash p : \tau$ *and* $\Psi \vdash p : \tau \rightsquigarrow p'$ *then* $|\Psi| \vdash p' : |\tau|$

**Proof:**
By definition, $p$ is of the form $\mathtt{letrec}\, d\, \mathtt{in}\, e$.
By assumption:
$\Psi, \Psi(d) \vdash d$ **ok**
$\Psi, \Psi(d) \vdash d \rightsquigarrow d_h\, \mathtt{in}\, d'$
$\Psi, \Psi(d); \bullet; \bullet \vdash e : \tau$ **exp**
$\Psi, \Psi(d); \bullet; \bullet \vdash e : \tau \rightsquigarrow d_e\, \mathtt{in}\, e'$
By lemma 25:
$|\Psi(d)| = \Psi(d')$
So by definition:
$|\Psi, \Psi(d)|$
$= |\Psi|, |\Psi(d)|$
$= |\Psi|, \Psi(d')$
By lemma 27:
$|\Psi, \Psi(d)|, \Psi(d_h) \vdash d_h, d'$ **ok**
And by lemma 13:
$|\Psi, \Psi(d)|, \Psi(d_e); \bullet; \bullet \vdash e' : |\tau|$ **exp**

So by the above argument:
$|\Psi|, \Psi(d'), \Psi(d_h) \vdash d_h, d'$ **ok**
and$|\Psi|, \Psi(d'), \Psi(d_e); \bullet; \bullet \vdash e' : |\tau|$ **exp**
By weakening:
$|\Psi|, \Psi(d'), \Psi(d_h), \Psi(d_e) \vdash d_h, d'$ **ok**
and$|\Psi|, \Psi(d'), \Psi(d_h), \Psi(d_e); \bullet; \bullet \vdash e' : |\tau|$ **exp**
So by the program rule :
$|\Psi| \vdash \texttt{letrec}\, d', d_h, d_e\, \texttt{in}\, e' : |\tau|$

$\blacksquare$

# Chapter 7

# TILTAL

Closure converted **LIL** code is sufficiently low-level that it is practical to translate it directly into a typed assembly language. In this chapter I present such a language, and define a simple translation from **LIL** into it. I call this assembly language TILT Typed Assembly Language (**TILTAL**), in reference to the original typed assembly language (**TAL**) [MWCG98]. The overall structure of **TILTAL** is very similar to the stack version of **TAL** [MCGW02].

The presentation of the **TILTAL** 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 targets the **TALx86** infrastructure [MCG$^{+}$99] which is significantly different in many ways. I will discuss the **TALx86** type system in more detail in the implementation chapters.

## 7.1 TILTAL overview

I begin by introducing the **TILTAL** language in general terms and describing the typing judgements which define its static semantics. The syntax for the language is described in figure 7.1. The kind and type levels are almost entirely the same as that of the **LIL** and are mostly elided. Constructs for describing the type of stacks and a kind $ST$ classifying them are added. Code function types 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 $ns_{64}$.

A table describing the typing judgements for **TILTAL** is given in figure 7.2. The complete list of inference rules for these judgements can be found in appendix C. Most **TILTAL** terms are judged well-typed with respect to a heap context $\Psi$ which binds labels at closed types; a constructor/kind context $\Delta$ which binds kind variables and binds constructor variables at kinds; and a register file type $\Gamma$ which describes the state of the abstract machine by mapping register names to types.

In the the next several sections, I will introduce the important syntactic classes in **TILTAL** and briefly describe their use. For the most part, readers familiar with the existing literature on typed assembly language will find nothing substantially different.

### 7.1.1 Stacks

The treatment of stacks and stack types in **TILTAL** is much the same as in previous work [MCGW02]. I give a brief introduction to the syntax and the concepts here.

$$
\begin{array}{lll}
k & ::= & \ldots \mid ST \\
c, \tau, \phi, \sigma & ::= & \ldots \boxed{\texttt{delete} \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 \\
\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}_{(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{brdyn}\, \mathbf{r}, sv_1, sv_2 \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{:}\tau_1 \mapsto hval_1, \ldots, \ell_n{:}\tau_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) \\
 & & \\
\Psi & ::= & \bullet \mid \Psi, \ell{:}\tau \\
\Delta & ::= & \bullet \mid \Delta, j \mid \Delta, \alpha{:}\kappa \\
\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\}
\end{array}
$$

**Figure 7.1: TILTAL** syntax

| **TILTAL** typing judgements | |
|---|---|
| Kinds | $\Delta \vdash \kappa$ **ok** |
| Constructors | $\Delta \vdash c : \kappa$ |
| Equivalence | $\Delta \vdash c \equiv c' : \kappa$ |
| Coercions | $\Delta \vdash q : \tau \Rightarrow \tau'$ |
| Stacks | $\Psi; \Delta \vdash s : \sigma$ |
| 64-bit values | $\Psi; \Delta \vdash l : \phi$ |
| 32-bit values | $\Psi; \Delta \vdash w : \tau$ |
| 64-bit operands | $\Psi; \Delta; \Gamma \vdash fv : \phi$ |
| 32-bit operands | $\Psi; \Delta; \Gamma \vdash sv : \tau$ |
| Instructions | $\Psi; \Delta; \Gamma \vdash i \Rightarrow \Gamma'$ |
| Instruction Sequences | $\Psi; \Delta; \Gamma \vdash I : \tau$ |
| Heap values | $\Psi\Delta; \Gamma \vdash hval : \tau$ **hval** |
| Heaps | $\vdash H : \Psi$ |
| Well-formed Register File | $\Psi \vdash R : \Gamma$ |
| Well-formed Program | $\vdash (H, R, I)$ **ok** |

**Figure 7.2:** Judgements defining well-typedness of **TILTAL** programs

Stack types and values in **TILTAL** 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. At the value level a stack is either empty ($\epsilon$), or a value pushed onto a stack ($w \triangleright_{32} s$ or $l \triangleright_{64} s$). Stacks in **TILTAL** 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 $\sigma$ is used for the type of stacks.

Stack types are built up from empty stacks using stack addition and composition. The type $\epsilon$ describes an empty stack, the type $\tau \triangleright_{32} \sigma$ describes a stack described by $\sigma$ with a 32 bit value described by $\tau$ pushed on top, and similarly for $\phi \triangleright_{64} \sigma$ when $\phi$ describes a 64 bit value. In this way, stack types closely parallel the structure of actual stacks. In addition to these constructs, stacks types may also be constructed by concatenating two stack types. Such a compound stack, written $\sigma_1 \circ \sigma_2$, is equivalent to the stack obtained by prepending all of $\sigma_1$ to $\sigma_2$. Finally, the type $\mathtt{sptr}(\sigma)$ describes a pointer to a stack described by $\sigma$. This facility is used to compile exception handlers, which must maintain pointers into the stack.

For syntactic convenience, I define several operations on stack types for use in the static semantics. I define stack type lookup operations $\sigma[i]_{32}$ and $\sigma[i]_{64}$ which find the appropriately size type in the stack $\sigma$ located $i$ words from the top of the stack.

**Definition 1 (Stack type 32 bit subscript)**

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

**Definition 2 (Stack type 64 bit subscript)**

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

In a similar manner, I define stack type updates $(\sigma)[i]_{32} \leftarrow \tau$ and $(\sigma)[i]_{64} \leftarrow \phi$ which replace the type at word offset $i$ in $\sigma$ with $\tau$ or $\phi$ respectively.

**Definition 3 (Stack type 32 bit update)**

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

**Definition 4 (Stack type 64 bit update)**

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

Finally, I define a size operation on stack types in the obvious manner.

**Definition 5 (Stack type size)**

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

### 7.1.2 Values

Small values in **TILTAL** are syntactically divided into 32 and 64-bit sizes, written with the meta-variables $w$ and $l$ respectively. The types of 32-bit values have kind $\mathrm{T}_{32}$ and are written using the meta-variable $\tau$, whereas the types of 64-bit values have kind $\mathrm{T}_{64}$ and are written with the meta-variable $\phi$. 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. 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 $\Psi$ maps labels to the types of the values to which they point.

### 7.1.3 Operands: Registers and stack slots

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

There are three sorts of registers in **TILTAL**: 32-bit registers, 64-bit registers, and the stack register. By convention, I reserve one of the 32-bit registers for the exception record (as described in section 8.1.2). Registers are used to hold the operands and intermediate values, and to pass arguments to functions. The registers given in the version of **TILTAL** described here are not specialized to the x86 platform, which has a very idiosyncratic register architecture.

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 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.

### 7.1.4 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.

The typing rules for instructions are somewhat different from the usual sorts of typing rules in that instead of ascribing a type to an instruction, they describe the effect that the instruction has on the type of the state. The judgement $\Psi; \Delta; \Gamma \vdash i \Rightarrow \Gamma'$ indicates that the instruction $i$, when executed in a state described by a register file type $\Gamma$, will result in a state described by $\Gamma'$. Note that instructions do not change the type of values in the heap, nor the set of free types.

In addition to standard instructions, there are several type instructions which have no runtime 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. Note that I syntactically segregate the type instructions from the ordinary instructions, and give typing rules for them only as elements of code sequences. This simplifies things greatly, since typing the refinement instructions individually would presumably require the instruction typing judgement to return a substitution with all of the related complications.

Instruction sequences are a sequence of instructions and type instructions, terminated by either `ret`, `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.

### 7.1.5 Heap values, heaps, and register files

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 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.

Register files are mappings from register names to values of the appropriate size and type for the given register. For example, register files map 32 bit register names to 32 bit word values. Register files are classified by register file types $\Gamma$.

### 7.1.6 Programs

A **TILTAL** program is a triple consisting of a heap $H$, a register file $R$, and an instruction sequence $I$. Execution of **TILTAL** program is defined by transitions between **TILTAL** programs. The complete dynamic semantics for **TILTAL** is given in appendix C.

### 7.1.7 Typing contexts

The form of heap contexts $\Psi$ and constructor contexts $\Delta$ are essentially unchanged from the **LIL**. As before, heap contexts give types to labels, and constructor contexts bind kind variables, and bind constructor variables at their kinds.

The term level context from the **LIL** is replaced by a register file type which describes the type of the registers and stack of the **TILTAL** abstract machine. The type of the register file (and consequently the stack as well) is given by a register file type $\Gamma$ which maps each register to the type of its contents. In addition to being used as part of the static semantics, register file types are included into the type level to describe the pre-conditions of code segments. The type $\Gamma \rightarrow 0$ describes a code segment which expects a register file described by $\Gamma$.

I use the notation $\Gamma(\mathbf{r})$ to indicate the type assigned by $\Gamma$ to a register $\mathbf{r}$, and similarly for float registers $\mathbf{f}$ and the stack register $\mathbf{sp}$. I use the notation $\Gamma\{\mathbf{r}{:}\tau\}$ to indicate the register file type obtained by replacing the type of $\mathbf{r}$ in $\Gamma$ with $\tau$, and analogously for float registers and stack registers.

## 7.2 TILTAL derived forms

### 7.2.1 Partial instruction sequences

In order to allow **TILTAL** expressions to be built up somewhat compositionally during the translation of **LIL** code, I define a derived notion of partial instruction sequences.

**Definition 6 (Partial instruction sequence)**
*A partial instruction sequence $S$ is a series of ordinary (non-refining) instructions not terminated by a jump, return or halt. That is,*

$$S ::= \epsilon \mid i; S$$

I define a judgement defining well-typedness as a partial instruction sequence: $\Psi; \Delta; \Gamma \vdash S \Rightarrow \Gamma'$ with inference rules as follows:

$$\frac{}{\Psi; \Delta; \Gamma \vdash \epsilon \Rightarrow \Gamma}$$

$$\frac{\Psi; \Delta; \Gamma \vdash i \Rightarrow \Gamma' \quad \Psi; \Delta; \Gamma' \vdash S \Rightarrow \Gamma''}{\Psi; \Delta; \Gamma \vdash i; S \Rightarrow \Gamma''}$$

**Definition 7 (Partial instruction sequence composition)**
*A partial instruction sequence $S$ can be composed with a second partial instruction sequence $S'$ to form a new instruction sequence $S; S'$ as follows:*

$$\begin{aligned} \epsilon; S' &\stackrel{\text{def}}{=} S' \\ (i; S); S' &\stackrel{\text{def}}{=} i; (S; S') \end{aligned}$$

The result of composing well-formed partial instruction sequences is well-formed.

**Lemma 28 (Soundness of partial instruction sequence composition)**
*If $\Psi; \Delta; \Gamma \vdash S \Rightarrow \Gamma'$ and $\Psi; \Delta; \Gamma' \vdash S' \Rightarrow \Gamma'$ then $\Psi; \Delta; \Gamma \vdash S; S' \Rightarrow \Gamma'$*

**Proof:** By induction on $S$. ∎

**Definition 8 (Partial instruction sequence completion)**
*A partial instruction sequence $S$ can be completed with an instruction sequence $I$ to form a new instruction sequence $S; I$ as follows:*

$$\begin{aligned} \epsilon; I &\stackrel{\text{def}}{=} I \\ (i; S); I &\stackrel{\text{def}}{=} i; (S; I) \end{aligned}$$

The result of completing a well-formed partial instruction sequence with a well-formed instruction sequence is itself well-formed: in this sense, partial instruction sequence completion is sound.

**Lemma 29 (Soundness of partial instruction sequence completion)**
*If $\Psi; \Delta; \Gamma \vdash S \Rightarrow \Gamma'$ and $\Psi; \Delta; \Gamma' \vdash I$ **ok** then $\Psi; \Delta; \Gamma \vdash S; I$ **ok***

**Proof:** By induction on $S$. ∎

### 7.2.2 Heap fragments

For convenience in the **LIL** to **TILTAL** translation, I also define a notion of partial, or open heaps, called heap fragments. Heap fragments are simply incomplete heaps, with their own derived typing judgement. Translations of **LIL** terms generally produce heap fragments in addition to **TILTAL** terms, reflecting the fact that the translation may place items in the static data segment.

**Definition 9 (Heap fragment)**
*A heap fragment $F$ is a syntactic heap, which may have open labels.*

I define a judgement defining well-typedness as a heap fragment of the form $\Psi \vdash F : \Psi'$.

$$\frac{\vdash \Psi, \Psi' \ \textbf{ok} \quad \Psi, \Psi' \vdash \mathit{hval}_i : \Psi'(\ell_i)\ \textbf{hval}}{\Psi \vdash \{\ell_1{:}\tau_1 \mapsto \mathit{hval}_1, \ldots, \ell_n{:}\tau_n \mapsto \mathit{hval}_n\} : \Psi'}$$

Two heap fragments $F_1$ and $F_2$ can be combined to form a new heap fragment $F_1; F_2$.

**Definition 10 (Heap fragment combination)**

$$\begin{aligned}&\{\ell_1{:}\tau_1 \mapsto \mathit{hval}_1, \ldots, \ell_n{:}\tau_n \mapsto \mathit{hval}_n\}; \{\ell_{n+1}{:}\tau_{n+1} \mapsto \mathit{hval}_{n+1}, \ldots, \ell_m{:}\tau_m \mapsto \mathit{hval}_m\}\\ &\stackrel{\text{def}}{=} \{\ell_1{:}\tau_1 \mapsto \mathit{hval}_1, \ldots, \ell_n{:}\tau_n \mapsto \mathit{hval}_n, \ell_{n+1}{:}\tau_{n+1} \mapsto \mathit{hval}_{n+1}, \ldots, \ell_m{:}\tau_m \mapsto \mathit{hval}_m\}\end{aligned}$$

The result of combining two well-formed heap fragments with disjoint labels is a well-formed heap-fragment.

**Lemma 30 (Soundness of heap fragment combination)**
*If $\Psi \vdash F_1 : \Psi_1$ and $\Psi \vdash F_2 : \Psi_2$ and the domains of $F_1$ and $F_@$ are disjoint, then $\Psi \vdash F_1; F_2 : \Psi_1, \Psi_2$*

**Proof:** By induction on $F_1$.

■

A heap fragment $F$ can be added to a heap $H$ to produce a new heap $F + H$.

**Definition 11 (Heap fragment incorporation)**

$$\begin{aligned}&\{\ell_1{:}\tau_1 \mapsto \mathit{hval}_1, \ldots, \ell_n{:}\tau_n \mapsto \mathit{hval}_n\} + \{\ell_{n+1}{:}\tau_{n+1} \mapsto \mathit{hval}_{n+1}, \ldots, \ell_m{:}\tau_m \mapsto \mathit{hval}_m\}\\ &\stackrel{\text{def}}{=} \{\ell_1{:}\tau_1 \mapsto \mathit{hval}_1, \ldots, \ell_n{:}\tau_n \mapsto \mathit{hval}_n, \ell_{n+1}{:}\tau_{n+1} \mapsto \mathit{hval}_{n+1}, \ldots, \ell_m{:}\tau_m \mapsto \mathit{hval}_m\}\end{aligned}$$

The result of incorporating a well-formed heap fragment with a well-formed heap with disjoint labels is well-formed.

**Lemma 31 (Soundness of heap fragment incorporation)**
*If $\vdash H : \Psi$ and $\Psi \vdash F : \Psi'$ and the domains of $H$ and $F$ are disjoint, then $\vdash H + F : \Psi, \Psi'$*

**Proof:** By induction on $F$.

■

# Chapter 8

# The LIL to TILTAL translation

The final stage of compilation translates **LIL** programs to **TILTAL** 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. 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 language design approach taken in **TILTAL** is not substantially different in nature from the that previously mapped out by Morrisett, et al. [MWCG98, MCGW02] (though the translation methodology is quite different). The actual code generation is not unusual except in that it preserves type information.

This chapter gives a detailed presentation of the translation of **LIL** programs into **TILTAL** programs, and proves the soundness of this translation with respect to an abstract model of register allocation.

## 8.1 The constructor level

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

### 8.1.1 Kinds

Every **LIL** kind is a **TILTAL** kind, and since no change is made in the constructor translation that changes kinds, the translation on kinds is the identity. In general, I treat every **LIL** kind as its own translation.

### 8.1.2 The constructor translation

The complete constructor translation is given in figure 8.1. Note that the $\rightarrow$ type constructor has no translation, since the translation is only defined on closure-converted programs. This could be made explicit by giving a syntactic variant of the **LIL** for closure converted programs and defining the closure conversion translation from chapter 6 as a translation between programs in the original **LIL** and programs in the new syntactic variant. However, it is much simpler to consider each variant as a refinement of a broader syntactic class. In general, throughout the translation I will

$$
\begin{array}{ll}
|\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| \\
|\,\mathtt{inj}_i^k\, c| & \stackrel{\text{def}}{=} \mathtt{inj}_i^k\,|c| \\
|\,\mathtt{case}(c, [\alpha_1.c_1, \ldots, \alpha_n.c_n])| & \stackrel{\text{def}}{=} \mathtt{case}(|c|, [\alpha_1.|c_1|, \ldots, \alpha_n.|c_n|]) \\
|\langle c1, c2\rangle| & \stackrel{\text{def}}{=} \langle |c1|, |c2| \rangle \\
|\,\mathtt{fold}_{\mu j.k}\, c| & \stackrel{\text{def}}{=} \mathtt{fold}_{\mu j.k}\,|c| \\
|\,\mathtt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}\,\mathtt{in}\, c)| & \stackrel{\text{def}}{=} \mathtt{pr}(j, \alpha{:}\kappa, \rho{:}(j \to \kappa'), \iota{:}\,\mathtt{in}\,|c|) \\
|\mathtt{Float}| & \stackrel{\text{def}}{=} \mathtt{Float} \\
|\mathtt{Int}| & \stackrel{\text{def}}{=} \mathtt{Int} \\
|\,\mathtt{Boxed}\,| & \stackrel{\text{def}}{=} \mathtt{Boxed} \\
|\mathtt{Void}| & \stackrel{\text{def}}{=} \mathtt{Void} \\
|\times| & \stackrel{\text{def}}{=} \times \\
|\to| & \stackrel{\text{def}}{=} \text{UNDEFINED} \\
|\,\mathtt{Code}\,| & \stackrel{\text{def}}{=} \\
\multicolumn{2}{l}{\quad \lambda(\alpha_{32}{:}\mathrm{T}_{32}\mathtt{list}, \alpha_{64}{:}\mathrm{T}_{64}\mathtt{list}, \alpha_r{:}\mathrm{T}_{32}).\forall[\rho_1{:}ST, \rho_2{:}ST].} \\
\multicolumn{2}{l}{\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{Exnptr}(\rho_2), \mathbf{r}_{\mathtt{t}}{:}ns_{32},} \\
\multicolumn{2}{l}{\qquad\quad \mathbf{sp}{:}\,\mathbf{cont}(|\alpha_{32} \mathbin{@_{32}} \alpha_{64} \mathbin{@_{64}} \epsilon|)(\rho_1)(\rho_2)(\alpha_r) \triangleright_{32} (\alpha_{32} \mathbin{@_{32}} (\alpha_{64} \mathbin{@_{64}} \rho_1 \circ \rho_2))\} \to 0} \\
|c[\kappa]| & \stackrel{\text{def}}{=} |c|[\kappa] \\
|\Lambda j.c| & \stackrel{\text{def}}{=} \Lambda j.|c| \\
|\exists| & \stackrel{\text{def}}{=} \exists \\
|\forall| & \stackrel{\text{def}}{=} \forall \\
|\,\mathtt{Rec}\,| & \stackrel{\text{def}}{=} \mathtt{Rec} \\
|\bigvee| & \stackrel{\text{def}}{=} \bigvee \\
|\mathtt{Array}_{32}| & \stackrel{\text{def}}{=} \mathtt{Array}_{32} \\
|\mathtt{Array}_{64}| & \stackrel{\text{def}}{=} \mathtt{Array}_{64} \\
|\,\mathtt{Dyntag}\,| & \stackrel{\text{def}}{=} \mathtt{Dyntag} \\
|\,\mathtt{Dyn}\,| & \stackrel{\text{def}}{=} \mathtt{Dyn} \\
|\,\mathtt{Tag}\,| & \stackrel{\text{def}}{=} \mathtt{Tag}
\end{array}
$$

**Figure 8.1:** The constructor translation

assume that the **LIL** programs under consideration are of the closure-converted variant and will not remark upon this further.

The only interesting work of the constructor translation is to translate the type of code functions into code sequence types of the form $\Gamma \to 0$. This can be thought of as replacing all uses of the `Code` constant by a closed defined form of the same kind. This is especially convenient, since the soundness of the translation follows almost directly after showing the well-formedness of this definition, since any well-formedness derivation of a **LIL** constructor can be turned into a well-formedness derivation of its translation by replacing all uses of the `Code` axiom with uses of the appropriately weakened well-formedness derivation for the definition.

### The translation of the `Code` type

In order to give the appropriate definition for the `Code` type, it is first necessary to discuss the calling conventions used by the translation, since these must be encoded in the types of code sequences. For simplicity in this translation, I use a so-called "caller pops" calling convention in which a called function is expected to return the stack with the space allocated for its in-arguments intact. This convention is not especially convenient for doing tail-call elimination: consequently the actual implementation uses a "callee-pops" calling convention in which a called function is expected to de-allocate the space allocated for its in-arguments before returning. The formal translation uses the caller-pops convention for simplicity.

The treatment of exception handlers in the translation is slightly different from that used by Morrisett et al. [MCGW02] in which two registers are dedicated to the exception handling mechanism. To correspond more closely with the current **TILT** implementation, I reserve a single register to hold the current exception frame. Exception frames contain the address of the current handler, and a pointer to the handler's stack. Old exception frames are stored on the top of the handler stack.

**Definition 12 (Exnptr and Exnhndler)**

$$
\begin{array}{lcl}
\mathbf{Exnhndler} & \stackrel{\text{def}}{=} & \lambda(\rho{:}ST). \\
& & \quad \forall[].\{\mathbf{r}_1{:}\,\mathtt{Dyn}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{r}_e{:}ns_{32}, \mathbf{sp}{:}\rho, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}\} \to 0 \\
\mathbf{Exnptr} & \stackrel{\text{def}}{=} & \lambda(\rho{:}ST). \times[\mathbf{Exnhndler}(\rho), \rho]
\end{array}
$$

The type of the exception handler itself is parameterized by the type of the handler stack. Notice that the handler expects an exception packet in $\mathbf{r}_1$.

Code function types in **TILTAL** must also encode the type of a function's continuation in the form of a return address type. I define a type $\mathbf{cont} : ST \to ST \to \mathrm{T}_{32}$ which describes the machine state expected by the calling code upon return. Continuation types are parameterized over the number of words of in-argument space on the stack, the return type, the remainder of the stack above the next exception handler, and the remainder of the stack below the next handler. 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.

**Definition 13 (cont)**

$$\mathbf{cont} \stackrel{\text{def}}{=} \lambda(\alpha_i{:}\mathtt{nat}, \rho_1{:}ST, \rho_2{:}ST, \alpha_{\mathbf{ret}}{:}\mathrm{T}_{32}).$$
$$\{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\rho_2), \mathbf{r_t}{:}\alpha_{\mathbf{ret}},$$
$$\mathbf{sp}{:}{ns_{32}}^{\alpha_i} \circ \rho_1 \circ \rho_2\} \to 0$$

The translation of the `Code` type using these auxiliary definitions is given in definition 14 below. Code function types take three arguments corresponding to a list of 32 bit arguments, a list of 64 bit arguments, and a return type. In addition the new code sequence is polymorphic over two stack variables: $\rho_1$ and $\rho_2$. In the translation of a function type, the second stack variable, $\rho_2$, classifies the stack tail expected by the current exception handler, while the first stack variable, $\rho_2$, classifies the stack between the arguments and the last handler. Functions expect all of their arguments on the stack, and the exception frame in register $\mathbf{r}_e$. The top most item on the stack upon entry to the function is the address of the code to which the function should return (that is, the function's continuation). Function return values are returned in register $\mathbf{r_t}$.

**Definition 14**

$$|\,\mathtt{Code}\,| \stackrel{\text{def}}{=} \lambda(\alpha_{32}{:}\mathrm{T}_{32}\mathtt{list}, \alpha_{64}{:}\mathrm{T}_{64}\mathtt{list}, \alpha_r{:}\mathrm{T}_{32}).\forall[\rho_1{:}ST, \rho_2{:}ST].$$
$$\{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\rho_2), \mathbf{r_t}{:}ns_{32},$$
$$\mathbf{sp}{:}\,\mathbf{cont}(|\alpha_{32} \mathbin{@_{\mathsf{32}}} \alpha_{64} \mathbin{@_{\mathsf{64}}} \epsilon|)(\rho_1)(\rho_2)(\alpha_r) \triangleright_{32} (\alpha_{32} \ \ @_{\mathsf{32}} \ \ (\alpha_{64} \ \ @_{\mathsf{64}} \ \ \rho_1 \circ \rho_2))\} \to 0$$

The construction of the stack type in definition 14 uses defined operators $@_{\mathsf{32}}$ and $@_{\mathsf{64}}$ which prepend lists of 32 bit and 64 bit types (respectively) to stack types. It is straightforward to define these as object level recursors within the type system.

**Definition 15 ($c \ \ @_{\mathsf{32}} \ \ \sigma$)**
$c \ \ @_{\mathsf{32}} \ \ \sigma = f(c)\sigma$ *where*

$$\mathbf{f}\,{:}\mathrm{T}_{32}\mathtt{list} \to ST \to ST \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{32}\mathtt{list}).\lambda(\rho{:}ST).\,\mathtt{pr}(j, \alpha{:}1 + \mathrm{T}_{32} \times j, \rho{:}(j \to ST),\ \mathtt{in}$$
$$\mathtt{case}\ (\alpha)$$
$$\mathtt{inj}_1 \star \Rightarrow \sigma$$
$$\mathtt{inj}_2\, \beta \Rightarrow ns_{32} \triangleright_{32} (\rho\beta))$$

**Definition 16 ($c \ \ @_{\mathsf{64}} \ \ \sigma$)**
$c \ \ @_{\mathsf{64}} \ \ \sigma = f(c)\sigma$ *where*

$$\mathbf{f}\,{:}\mathrm{T}_{64}\mathtt{list} \to ST \to ST \stackrel{\text{def}}{=} \lambda(\alpha{:}\mathrm{T}_{64}\mathtt{list}).\lambda(\rho{:}ST).\,\mathtt{pr}(j, \alpha{:}1 + \mathrm{T}_{32} \times j, \rho{:}(j \to ST),\ \mathtt{in}$$
$$\mathtt{case}\ (\alpha)$$
$$\mathtt{inj}_1 \star \Rightarrow \sigma$$
$$\mathtt{inj}_2\, \beta \Rightarrow ns_{32} \triangleright_{64} (\rho\beta))$$

The definition also refers to the length function for stacks $|\cdot|{:}ST \to \mathtt{nat}$. It is straightforward to define this using primitive recursion in the same fashion as the iterated stack type and the list append functions.

### 8.1.3 Soundness of the type translation

**Lemma 32 (Well-formedness of the Code definition)**
*If* $\vdash \Delta$ **ok** *then* $\Delta \vdash \mathbf{Code} : \mathrm{T}_{32}\mathtt{list} \to \mathrm{T}_{64}\mathtt{list} \to \mathrm{T}_{32} \to \mathrm{T}_{32}$.

**Proof:** By construction, $\vdash \mathbf{Code} : \mathrm{T}_{32}\mathtt{list} \to \mathrm{T}_{64}\mathtt{list} \to \mathrm{T}_{32} \to \mathrm{T}_{32}$. The result then follows by repeated weakening.

■

**Theorem 15 (Soundness of the constructor translation)**
*For a* **LIL** *constructor* $c$*, if* $\Delta \vdash c : \kappa$ *then* $\Delta \vdash |c| : \kappa$.

**Proof:** By construction. Every derivation $D$ of $\Delta \vdash c : \kappa$ can be turned into a derivation of $\Delta \vdash |c| : \kappa$ by replacing every use of the `Code` axiom with a derivation for **Code** obtained via lemma 32.

■

**Theorem 16 (The translation respects substitution)**
$|c|[|c'|/\alpha] = |c[c'/\alpha]|$

**Proof:** The code definition (**Code**) and the `Code` primitive are both closed, so

$$|\,\mathtt{Code}\,|[|c'|/\alpha] = \mathbf{Code}[|c'|/\alpha] = \mathbf{Code} = |\,\mathtt{Code}\,| = |\,\mathtt{Code}[c'/\alpha]|$$

All other constructors remain unchanged by the translation, so the proof follows trivially.

■

**Theorem 17 (The translation respects equivalence)**
*If* $\Delta \vdash c \equiv c' : \kappa$ *then* $\Delta \vdash |c| \equiv |c'| : \kappa$

**Proof:** By construction. Every derivation of $\Delta \vdash c \equiv c' : \kappa$ can be turned into a derivation of $\Delta \vdash |c| \equiv |c'| : \kappa$ by replacing every use of the reflexivity axiom on `Code` with a use of the reflexivity axiom on **Code**.

■

## 8.2 The term level: preliminaries

Most of the work of the translation from **LIL** to **TILTAL** takes place at the level of terms. The next few sections will introduce some of the key concepts used in the translation.

### 8.2.1 Register and stack slot allocation

One of the key issues in translating from a variable binding/substitution based language to a register transfer style language is how to efficiently manage a fixed set of registers. This is the problem of register allocation. For the purposes of the formal translation, I choose to leave the specifics of register and stack slot allocation abstract. The problem of register allocation has been studied extensively, and this dissertation does not add anything substantially new to the discussion. In practice, standard techniques can be applied to determine a suitable mapping and the translation uses this information abstractly. This has the advantage of essentially making the translation parametric over the choice of allocation methods. Nonetheless, I wish to be able to show that the translation as I define it is sound. Consequently, I will place certain typing requirements on the allocation method.

The term level translation is defined with respect to a notion of an abstract allocator, which I generally write as $\mathcal{A}$. An allocator is an object that is responsible for mapping program variables (both 32 and 64 bit) to registers and stack slots of the appropriate size. In order to avoid relying on any particular register allocation technology, I generally remain agnostic as to the internal data structures of the allocator, defining only a set of operations which any allocator must support.

**Definition 17**
*An allocator is an object $\mathcal{A}$ with the following associated operations:*

- *For every 32 bit variable $x$, $\mathcal{A}(x) = \mathbf{r}$ or $\mathcal{A}(x) = \mathbf{sp}(i)$.*
- *For every 64 bit variable $x_f$, $\mathcal{A}(x_f) = \mathbf{f}$ or $\mathcal{A}(x_f) = \mathbf{sp}(i)$.*
- $\mathrm{frmsz}(\mathcal{A})$ *is a natural number.*
- *For every* **LIL** *typing context $\Gamma$ and stack type $\sigma$, $|\Gamma|^{\sigma}_{\mathcal{A}} = \Gamma_A$ for some register file type $\Gamma_A$.*

The most basic such operation is to give a location for a variable. For a 32 bit variable $x$, I write the location associated with $x$ by an allocator $\mathcal{A}$ as $\mathcal{A}(x)$. Where appropriate, I sometimes write $\mathcal{A}[x \to \mathbf{r}]$ when $\mathcal{A}(x) = \mathbf{r}$ or $\mathcal{A}[x \to \mathbf{sp}(i)]$ when $\mathcal{A}(x) = \mathbf{sp}(i)$. Similarly, for a 64 bit variable $x_{64}$, I write the location associated with $x_{64}$ by the allocator $\mathcal{A}$ as $\mathcal{A}(x_{64})$. Where appropriate, I sometimes write $\mathcal{A}[x_{64} \to \mathbf{f}]$ when $\mathcal{A}(x_{64}) = \mathbf{f}$ or $\mathcal{A}[x_{64} \to \mathbf{sp}(i)]$ when $\mathcal{A}(x_{64}) = \mathbf{sp}(i)$. It is perfectly reasonable (and likely) that an allocator will map multiple variables to the same registers and stack slots [1]

An allocator defines and manages the stack frame for a function. Consequently, I require that an allocator answer queries about the size (in 32 bit words) of the current frame: written $\mathrm{frmsz}(\mathcal{A})$.

An allocator is responsible for managing the stack and register resources used to implement a **LIL** program in the **TILTAL** abstract machine. At various points in the program, it is necessary to write out the type of the abstract machine. Consequently, the second operation required of an allocator is to be able to generate a register file type from a given typing context $\Gamma$. Note though that an allocator is only responsible for managing the current *frame* and the registers - it should be parametric with respect to the rest of the stack. With this in mind, I define the translation of a **LIL** typing context $\Gamma$ under an allocator $\mathcal{A}$ with respect to a stack tail $\sigma$ to be the register file $|\Gamma|^{\sigma}_{\mathcal{A}}$.

This completely defines an allocator: an object $\mathcal{A}$ which assigns locations to variables, has a defined frame size, and maps every context/stack type pair to a register file type. However, not every allocator is sufficient for the purposes of the translation, since there are many incoherent allocators which satisfy this definition. In order to state the soundness of the **LIL** to **TILTAL** translation, I define the notion of a *good allocator* that satisfies certain conditions.

**Definition 18 (Good allocator for a context)**
*Let $\Gamma_A$ be $|\Gamma|^{\sigma}_{\mathcal{A}}$, where $\Gamma$ is a context and $\sigma$ is a stack type, and let $\mathbf{r_t}$, $\mathbf{f_t}$, and $\mathbf{r}_e$ be designated machine registers. Then I say that an allocator $A$ is a good allocator for $\Gamma$ if:*

[1] While for semantic correctness it is to be hoped that this will only occur for variables whose live ranges do not overlap, this is *not* required by the translation so long as the typing requirements discussed below are fulfilled.

1. *For all $x$, $\mathcal{A}(x) \neq \mathbf{r}_e$ and $\mathcal{A}(x) \neq \mathbf{r_t}$ and for all $x_{64}$, $\mathcal{A}(x) \neq \mathbf{f_t}$.*

2. $\Gamma_A(\mathbf{sp}) = \sigma_f \circ \sigma$ *and* $\mathrm{frmsz}(\mathcal{A}) = |\sigma_f|$.

3. $|\bullet|^\sigma_\mathcal{A} = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}ns_{32}, \mathbf{r_t}{:}ns_{32}, \mathbf{sp}{:}\overline{ns_{32}}^{n} \circ \sigma\}$ *where* $n = \mathrm{frmsz}(\mathcal{A})$.

4. *If* $\Gamma = \Gamma_1, x{:}\tau, \Gamma_2$ *then:*

   (a) *$\mathcal{A}$ is a good allocator for $\Gamma_1, \Gamma_2$.*

   (b) *If* $\mathcal{A}(x) = \mathbf{r}$ *then* $|\Gamma|^\sigma_\mathcal{A} = |\Gamma_1, \Gamma_2|^\sigma_\mathcal{A}\{\mathbf{r}{:}|\tau|\}$

   (c) *If* $\mathcal{A}(x) = \mathbf{sp}(i)$ *then* $|\Gamma|^\sigma_\mathcal{A} = |\Gamma_1, \Gamma_2|^\sigma_\mathcal{A}\{\mathbf{sp}{:}(\sigma')[i]_{32} \leftarrow |\tau|\}$ *where* $|\Gamma_1, \Gamma_2|^\sigma_\mathcal{A}(\mathbf{sp}) = \sigma'$.

5. *If* $\Gamma = \Gamma_1, x_{64}{:}\phi, \Gamma_2$ *then:*

   (a) *$\mathcal{A}$ is a good allocator for $\Gamma_1, \Gamma_2$.*

   (b) *If* $\mathcal{A}(x_{64}) = \mathbf{f}$, $|\Gamma|^\sigma_\mathcal{A} = |\Gamma_1, \Gamma_2|^\sigma_\mathcal{A}\{\mathbf{f}{:}|\phi|\}$

   (c) *If* $\mathcal{A}(x) = \mathbf{sp}(i)$ *then* $|\Gamma|^\sigma_\mathcal{A} = |\Gamma_1, \Gamma_2|^\sigma_\mathcal{A}\{\mathbf{sp}{:}(\sigma')[i]_{64} \leftarrow |\phi|\}$ *where* $|\Gamma_1, \Gamma_2|^\sigma_\mathcal{A}(\mathbf{sp}) = \sigma'$.

The first three requirements ensure that the allocator doesn't interfere with registers and stack slots outside of its control, and that all unused registers and stack slots are given the nonsense type by the translation.

The special registers $\mathbf{r_t}$, $\mathbf{f_t}$ $\mathbf{r}_e$ are not available for general allocation. The first two are reserved for temporary computation and returned values, and the last for exception frames.

The second requirement of a good allocator is that the stack type in the machine state returned from the translation of a context must be an extension of the stack provided, and that the size of the extension must match the result of the frmsz() query.

The third requirement of a good allocator is that it not impose extraneous requirements on the state: that is, that the translation of an empty typing context is empty.

The important requirement to be a good allocator is that the locations returned from variable location queries must be coherent with the machine state obtained by translating a context containing the query variables. This constraint is expressed in the last two good allocator constraints.

There are two properties of interest that follow from these constraints. The first can be summarized informally as follows: if $\Gamma(x) = \tau$ then $|\Gamma|^\sigma_\mathcal{A}(\mathcal{A}(x)) = |\tau|$: that is, if the allocator assigns a variable from a typing context to a location, then the type assigned to that location in the translation of the context must be the translation of the variable type. More precisely, if $\Gamma_A = |\Gamma|^\sigma_\mathcal{A}$ then:

1. If for every $x$ such that $\Gamma = \Gamma_1, x{:}\tau, \Gamma_2$:

   (a) If $\mathcal{A}(x) = \mathbf{r}$, then $\Gamma_A(\mathbf{r}) = |\tau|$.

   (b) If $\mathcal{A}(x) = \mathbf{sp}(i)$ and $\Gamma_A(\mathbf{sp}) = \sigma'$ then $\sigma'[i]_{32} = |\tau|$.

2. If for every $x$ such that $\Gamma = \Gamma_1, x_{64}{:}\phi, \Gamma_2$:

   (a) If $\mathcal{A}(x_{64}) = \mathbf{f}$, then $\Gamma_A(\mathbf{f}) = |\phi|$.

   (b) If $\mathcal{A}(x_{64}) = \mathbf{sp}(i)$ and $\Gamma_A(\mathbf{sp}) = \sigma'$ then $\sigma'[i]_{64} = |\phi|$.

The second property that follows from the definition is that the extension of a typing context by a single variable translates to the same machine state as the translation of the original context with the type of the location for the new variable updated with the appropriate type. Informally, $|\Gamma, x{:}\tau|^\sigma_\mathcal{A} = |\Gamma|^\sigma_\mathcal{A}\{\mathcal{A}(x){:}|\tau|\}$. More precisely:

1. If $\Gamma = \Gamma', x{:}\tau$ then:
   (a) If $\mathcal{A}(x) = \mathbf{r}$ then $|\Gamma|^\sigma_\mathcal{A} = |\Gamma'|^\sigma_\mathcal{A}\{\mathbf{r}{:}|\tau|\}$
   (b) If $\mathcal{A}(x) = \mathbf{sp}(i)$ then $|\Gamma|^\sigma_\mathcal{A} = |\Gamma'|^\sigma_\mathcal{A}\{\mathbf{sp}{:}(\sigma')[i]_{32} \leftarrow |\tau|\}$ where $|\Gamma'|^\sigma_\mathcal{A}(\mathbf{sp}) = \sigma'$.
2. If $\Gamma = \Gamma', x_{64}$ then:
   (a) If $\mathcal{A}(x_{64}) = \mathbf{f}$, $|\Gamma|^\sigma_\mathcal{A} = |\Gamma'|^\sigma_\mathcal{A}\{\mathbf{f}{:}|\phi|\}$
   (b) If $\mathcal{A}(x) = \mathbf{sp}(i)$ then $|\Gamma|^\sigma_\mathcal{A} = |\Gamma'|^\sigma_\mathcal{A}\{\mathbf{sp}{:}(\sigma')[i]_{64} \leftarrow |\phi|\}$ where $|\Gamma'|^\sigma_\mathcal{A}(\mathbf{sp}) = \sigma'$.

A third (indirect) consequence of the good allocator definition is that the translation of contexts commutes with substitution.

**Lemma 33 (Context translation substitution)**
*If $\mathcal{A}$ is a good allocator for $\Gamma$ then*

$$|\Gamma|^\sigma_\mathcal{A}[|c|/\alpha] = |\Gamma[c/\alpha]|^{\sigma[|c|/\alpha]}_\mathcal{A}$$

**Proof:** (By induction on $\Gamma$).

1. If $\Gamma = \bullet$ then

$$\begin{aligned}
&|\Gamma|^\sigma_\mathcal{A}[|c|/\alpha] = \\
&\quad \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}ns_{32}, \mathbf{r_t}{:}ns_{32}, \mathbf{sp}{:}\overline{ns_{32}}^{n} \circ \sigma\}[|c|/\alpha] = \\
&\quad \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}ns_{32}, \mathbf{r_t}{:}ns_{32}, \mathbf{sp}{:}\overline{ns_{32}}^{n} \circ \sigma[|c|/\alpha]\} = \\
&\quad |\Gamma[c/\alpha]|^{\sigma[|c|/\alpha]}_\mathcal{A}
\end{aligned}$$

2. If $\Gamma = \Gamma', x{:}\tau$ then by the good allocator assumption (using informal notation)

$$\begin{aligned}
&|\Gamma|^\sigma_\mathcal{A}[|c|/\alpha] = \text{(by definition)} \\
&\quad (|\Gamma'|^\sigma_\mathcal{A}\{\mathcal{A}(x){:}|\tau|\})[|c|/\alpha] = \text{(by the good allocator assumption)} \\
&\quad (|\Gamma'|^\sigma_\mathcal{A}[|c|/\alpha])\{\mathcal{A}(x){:}|\tau|[|c|/\alpha]\} = \text{(by definition)} \\
&\quad (|\Gamma'|^\sigma_\mathcal{A}[|c|/\alpha])\{\mathcal{A}(x){:}|\tau[c/\alpha]|\} = \text{(by lemma 16)} \\
&\quad |\Gamma'[c/\alpha]|^{\sigma[|c|/\alpha]}_\mathcal{A}\{\mathcal{A}(x){:}|\tau[c/\alpha]|\} = \text{(by induction)} \\
&\quad |\Gamma[c/\alpha]|^{\sigma[|c|/\alpha]}_\mathcal{A}
\end{aligned}$$

■

Being a good allocator is a very weak requirement on allocators. In particular, it does not guarantee that the choice of locations is semantically well-behaved. For example, the allocator which assigns every variable of a given type to the same location (regardless of liveness) is a perfectly good allocator by this definition: the fact that such an allocator is a poor choice practically is irrelevant from a type-soundness standpoint.

The notion of a good allocator for a context $\Gamma$ extends naturally to a notion of a good allocator for an expression.

**Definition 19 (Good allocator for an expression)**
*If $D$ is a derivation of $\Psi; \Delta; \Gamma; \mathcal{A} \vdash e : \tau$ then I say that an allocator $\mathcal{A}$ is a good allocator for $e$ if for all sub-derivations of $D$ of the form $\Psi; \Delta'; \Gamma' \vdash e' : \tau'$ (for some $\Delta', \Gamma', e', \tau'$), $\mathcal{A}$ is a good allocator for $\Gamma'$.*

Note that this implies that a good allocator for $e$ is a good allocator for all sub-expressions of $e$. Also note that this implies that $\mathcal{A}$ is a good allocator for $\Gamma$.

Since expressions also occur as sub-derivations of operations (in case expressions and handlers) it is necessary to define an analogous property for operations.

**Definition 20 (Good allocator for an operation)**
*If $D$ is a derivation of $\Psi; \Delta; \Gamma; \mathcal{A} \vdash opr : \tau\ \mathbf{opr}_{32}$ then I say that an allocator $\mathcal{A}$ is a good allocator for $opr$ if for all sub-derivations of $D$ of the form $\Psi; \Delta'; \Gamma' \vdash e' : \tau'$ (for some $\Delta', \Gamma', e', \tau'$), $\mathcal{A}$ is a good allocator for $\Gamma'$.*

Note that this implies that a good allocator for an expression $e$ is a good allocator for all the operations in $e$, and that a good allocator for an operation $opr$ is a good allocator for all expressions in $opr$.

### 8.2.2 Derived instructions

It is convenient in the course of the translation to make use of some derived instructions not provided in the core instruction set but which can be defined as sequence of core instructions. In a complete implementation these instructions might be added as primitive.

The first defined instructions coerce registers and stack slots to the nonsense type.

**Definition 21 (Junk instructions)**

$$\begin{array}{lcl}
\mathbf{junk}\ r & \stackrel{\text{def}}{=} & \texttt{mov}\ r, ns_{32} \\
\mathbf{fjunk}\ f & \stackrel{\text{def}}{=} & \texttt{fmov}\ f, ns_{64} \\
\mathbf{sjunk}\ \mathbf{sp}(i) & \stackrel{\text{def}}{=} & \texttt{swrite}\ \mathbf{sp}(i), ns_{32} \\
\mathbf{fsjunk}\ \mathbf{sp}(i) & \stackrel{\text{def}}{=} & \texttt{fswrite}\ \mathbf{sp}(i), ns_{64}
\end{array}$$

For brevity, a single instruction is defined to coerce all registers to the nonsense type.

**Definition 22 (Register coercion)**

$$\begin{array}{lcl}
\mathbf{junkregs} & \stackrel{\text{def}}{=} & \mathbf{junk}\ \mathbf{r}_1; \\
& & \mathbf{junk}\ \mathbf{r}_2; \\
& & \mathbf{junk}\ \mathbf{f}_1; \\
& & \mathbf{junk}\ \mathbf{f}_2; \\
& & \epsilon
\end{array}$$

**Lemma 34 (Derived instructions)**
*For well-formed contexts $\Psi$, $\Delta$, and $\Gamma$, where $\Gamma(\mathbf{sp}) = \sigma$*

- $\Psi; \Delta; \Gamma \vdash \mathbf{junk}\ \mathbf{r} \Rightarrow \Gamma\{\mathbf{r}{:}ns_{32}\}$

- $\Psi; \Delta; \Gamma \vdash \textbf{fjunk } \mathbf{f} \Rightarrow \Gamma\{\mathbf{f}{:}ns_{64}\}$
- $\Psi; \Delta; \Gamma \vdash \textbf{sjunk } \mathbf{sp}(i) \Rightarrow \Gamma\{\mathbf{sp}{:}(\sigma)[i]_{32} \leftarrow ns_{32}\}$
- $\Psi; \Delta; \Gamma \vdash \textbf{fsjunk } \mathbf{sp}(i) \Rightarrow \Gamma\{\mathbf{sp}{:}(\sigma)[i]_{64} \leftarrow ns_{64}\}$
- $\Psi; \Delta; \Gamma \vdash \textbf{junkregs } \mathbf{r} \Rightarrow \Gamma\{\mathbf{r}_1{:}ns_{32}\}\{\mathbf{r}_2{:}ns_{32}\}\{\mathbf{f}_1{:}ns_{64}\}\{\mathbf{f}_2{:}ns_{64}\}$

**Proof:** By construction.

■

Finally, I give an inductive definition of an instruction to coerce a range of stack slots to the nonsense type.

**Definition 23 (Stack range coercion)**

$$\begin{array}{lll} \textbf{junkstack } n \ldots n & \stackrel{\text{def}}{=} & \textbf{sjunk}\,\mathbf{sp}(n); \\ & & \epsilon \\ \textbf{junkstack } n \ldots m \quad (n < m) & \stackrel{\text{def}}{=} & \textbf{sjunk}\,\mathbf{sp}(m); \\ & & \textbf{junkstack } n \ldots (m-1) \end{array}$$

**Lemma 35 (Stack range coercion)**
*For natural numbers $n$ and $m$ where $n < m$ and for well-formed contexts $\Psi$, $\Delta$, and $\Gamma$, where $\Gamma(\mathbf{sp}) = \sigma$:*

$$\Psi; \Delta; \Gamma \vdash \textbf{junkstack } n \ldots m \Rightarrow \Gamma\{\mathbf{sp}{:}\sigma'\}$$

*where $\sigma' = (((\sigma)[n]_{32} \leftarrow ns_{32}) \ldots)[m]_{32} \leftarrow ns_{32}$*

**Proof:** By induction on m. If $n = m$, then the derived instruction sequence is $\textbf{sjunk } \mathbf{sp}(m){:}\epsilon$, and the result follows directly by lemma 34. If $n < m$, then by induction, we get an instruction sequence which junks $\mathbf{sp}(n), \ldots, \mathbf{sp}(m-1)$. After appending on the additional $\textbf{sjunk } \mathbf{sp}(m)$ instruction, the result again follows by lemma 34.

■

In addition to these composite instructions, I also define a more general form of the move instruction which targets either registers or stack slots.

**Definition 24 (Arbitrary `mov` and `fmov`)**

$$\begin{array}{lll} \textbf{srmov } \mathbf{r}, sv & \stackrel{\text{def}}{=} & \texttt{mov } \mathbf{r}, sv \\ \textbf{srmov } \mathbf{sp}(i), sv & \stackrel{\text{def}}{=} & \texttt{swrite } \mathbf{sp}(i), sv \\ \textbf{srfmov } \mathbf{f}, fv & \stackrel{\text{def}}{=} & \texttt{mov } \mathbf{f}, fv \\ \textbf{srfmov } \mathbf{sp}(i), fv & \stackrel{\text{def}}{=} & \texttt{fswrite } \mathbf{sp}(i), fv \end{array}$$

**Lemma 36 (Arbitrary `mov`)**
*For a location $dest_{32}$, and for an operand $sv$ such that $\Psi; \Delta; \Gamma \vdash sv : \tau$*

- *If $dest_{32} = \mathbf{r}$ then $\Psi; \Delta; \Gamma \vdash \textbf{srmov } \mathbf{r}, sv \Rightarrow \Gamma\{\mathbf{r}{:}\tau\}$*
- *If $dest_{32} = \mathbf{sp}(i)$ and $\Gamma(\mathbf{sp}) = \sigma$ then $\Psi; \Delta; \Gamma \vdash \textbf{srmov } \mathbf{sp}(i), sv \Rightarrow \Gamma\{\mathbf{sp}{:}(\sigma)[i]_{32} \leftarrow \tau\}$*

**Proof:** By construction.

■

**Lemma 37 (Arbitrary fmov)**
*For a location $dest_{32}$, and for an operand $fv$ such that $\Psi; \Delta; \Gamma \vdash fv : \phi$*

- *If* $dest_{32} = \mathbf{f}$ *then* $\Psi; \Delta; \Gamma \vdash \textbf{fsrmov}\ \mathbf{f}, fv \Rightarrow \Gamma\{\mathbf{f}{:}\phi\}$
- *If* $dest_{32} = \mathbf{sp}(i)$ *and* $\Gamma(\mathbf{sp}) = \sigma$ *then* $\Psi; \Delta; \Gamma \vdash \textbf{fsrmov}\ \mathbf{sp}(i), fv \Rightarrow \Gamma\{\mathbf{sp}{:}(\sigma)[i]_{64} \leftarrow \phi\}$

**Proof:** By construction.

■

To simplify the translation (as well as eliminate un-necessary moves), I define register to register `mov` instructions which coerce the source register to $ns_{32}$ after the move whenever the registers are not aliases.

**Definition 25 (Move and junk)**

$$
\begin{array}{lll}
\textbf{movj}\ \mathbf{r}, \mathbf{r} & \stackrel{\text{def}}{=} & \epsilon \\
\textbf{movj}\ \mathbf{r}, \mathbf{r}'\ (\mathbf{r} \neq \mathbf{r}') & \stackrel{\text{def}}{=} & \texttt{mov}\ \mathbf{r}, \mathbf{r}'; \\
 & & \textbf{junk}\ \mathbf{r}' \\
\textbf{fmovj}\ \mathbf{f}, \mathbf{f} & \stackrel{\text{def}}{=} & \epsilon \\
\textbf{fmovj}\ \mathbf{f}, \mathbf{f}'\ (\mathbf{f} \neq \mathbf{f}') & \stackrel{\text{def}}{=} & \texttt{fmov}\ \mathbf{f}, \mathbf{f}'; \\
 & & \textbf{fjunk}\ \mathbf{f}'
\end{array}
$$

**Lemma 38 (32 bit move and junk)**
*For registers* $\mathbf{r}$ *and* $\mathbf{r}'$ $\Psi; \Delta; \Gamma \vdash \textbf{movj}\ \mathbf{r}, \mathbf{r}' \Rightarrow \Gamma\{\mathbf{r}'{:}ns_{32}\}\{\mathbf{r}{:}\Gamma(\mathbf{r}')\}$

**Proof:** By construction.

If the registers are aliases, then the instruction sequence is empty, and

$$\Gamma\{\mathbf{r}'{:}ns_{32}\}\{\mathbf{r}{:}\Gamma(\mathbf{r}')\} = \Gamma\{\mathbf{r}{:}ns_{32}\}\{\mathbf{r}{:}\Gamma(\mathbf{r})\} = \Gamma$$

If the registers are different, then `mov` and **junk** instructions are emitted, which update the register file type accordingly.

■

**Lemma 39 (64 bit move and junk)**
*For registers* $\mathbf{f}$ *and* $\mathbf{f}'$ $\Psi; \Delta; \Gamma \vdash \textbf{fmovj}\ \mathbf{f}, \mathbf{f}' \Rightarrow \Gamma\{\mathbf{r}'{:}ns_{64}\}\{\mathbf{f}{:}\Gamma(\mathbf{f}')\}$

**Proof:** By construction. If the registers are aliases, then the instruction sequence is empty, and $\Gamma = \Gamma\{\mathbf{f}'{:}ns_{64}\}\{\mathbf{f}{:}\Gamma(\mathbf{f}')\}$. If the registers are different, then `fmov` and **fjunk** instructions are emitted, which update the register file type accordingly.

■

For exception handlers, I define a code sequence for copying a stack segment. I begin by defining a code sequence **stackcopy** $(\sigma_1, \sigma_2)$ which copies the portion of the stack described by $\sigma_2$ intro pre-allocated space below the already copied $\sigma_1$. Note that **stackcopy** $(\sigma_1, \sigma_2)$ is only defined when $\sigma_1$ and $\sigma_2$ have well-defined sizes: that is, when they are composed solely of pushes and of compositions of stacks with well-defined sizes.

$$\begin{array}{lcl}
\textbf{stackcopy}\ (\sigma, \epsilon) & \stackrel{\text{def}}{=} & \epsilon \\
\textbf{stackcopy}\ (\sigma_1, \tau \triangleright_{32} \sigma_2) & \stackrel{\text{def}}{=} & \texttt{swrite}\ \textbf{sp}(|\sigma_1|), \textbf{sp}(|\sigma_1 \circ \tau \triangleright_{32} \sigma_2| + |\sigma_1|); \\
& & \textbf{stackcopy}\ (\sigma_1 \circ (\tau \triangleright_{32} \epsilon), \sigma_2) \\
\textbf{stackcopy}\ (\sigma_1, \phi \triangleright_{64} \sigma_2) & \stackrel{\text{def}}{=} & \texttt{fswrite}\ \textbf{sp}(|\sigma_1|), \textbf{sp}(|\sigma_1 \circ \phi \triangleright_{64} \sigma_2| + |\sigma_1|); \\
& & \textbf{stackcopy}\ (\sigma_1 \circ (\phi \triangleright_{64} \epsilon), \sigma_2)
\end{array}$$

**Lemma 40 (stackcopy)**
*For well-formed stack types $\sigma_1$, $\sigma_2$ and $\sigma$ such that $|\sigma_1|$ and $|\sigma_2|$ are well-defined:*

$$\Psi; \Delta; \Gamma\{\textbf{sp}{:}(\sigma_1 \circ {ns_{32}}^{|\sigma_2|}) \circ (\sigma_1 \circ \sigma_2) \circ \sigma\} \vdash \textbf{stackcopy}(\sigma_1, \sigma_2) \Rightarrow \Gamma\{\textbf{sp}{:}(\sigma_1 \circ \sigma_2) \circ (\sigma_1 \circ \sigma_2) \circ \sigma\}$$

**Proof:** By induction on $\sigma_2$.

- If $\sigma_2$ is empty, then $|\sigma_2| = 0$, so

$$(\sigma_1 \circ {ns_{32}}^{|\sigma_2|}) \circ (\sigma_1 \circ \sigma_2) \circ \sigma = \sigma_1 \circ \sigma_1 \circ \sigma$$

- If $\sigma_2 = \tau \triangleright_{32} \sigma_2'$ then:

  Let $\sigma_1' = \sigma_1 \circ (\tau \triangleright_{32} \epsilon)$

  By induction:
  $$\Psi; \Delta; \Gamma\{\textbf{sp}{:}(\sigma_1' \circ {ns_{32}}^{|\sigma_2'|}) \circ (\sigma_1' \circ \sigma_2') \circ \sigma\} \vdash \textbf{stackcopy}(\sigma_1', \sigma_2') \Rightarrow \Gamma\{\textbf{sp}{:}(\sigma_1' \circ \sigma_2') \circ (\sigma_1' \circ \sigma_2') \circ \sigma\}$$

  Note that $\sigma_1' \circ \sigma_2' = \sigma_1 \circ (\tau \triangleright_{32} \epsilon) \circ \sigma_2' = \sigma_1 \circ \tau \triangleright_{32} \sigma_2' = \sigma_1 \circ \sigma_2$

  So by the partial sequence instruction rule, it suffices to show that:
  $$\Psi; \Delta; \Gamma\{\textbf{sp}{:}(\sigma_1 \circ {ns_{32}}^{|\sigma_2|}) \circ (\sigma_1 \circ \sigma_2) \circ \sigma\} \vdash \texttt{swrite}\ \textbf{sp}(|\sigma_1|), \textbf{sp}(|\sigma_1 \circ \tau \triangleright_{32} \sigma_2'| + |\sigma_1|) \Rightarrow \Gamma\{\textbf{sp}{:}(\sigma_1 \circ (\tau \triangleright_{32} \epsilon) \circ {ns_{32}}^{|\sigma_2'|}) \circ (\sigma_1 \circ \sigma_2) \circ \sigma\}$$

  But note that $|\sigma_1 \circ {ns_{32}}^{|\sigma_2|} \circ \sigma_1| = |\sigma_1 \circ \tau \triangleright_{32} \sigma_2'| + |\sigma_1|$, and hence

  $$((\sigma_1 \circ {ns_{32}}^{|\sigma_2|}) \circ (\sigma_1 \circ \sigma_2) \circ \sigma)[|\sigma_1 \circ \tau \triangleright_{32} \sigma_2'| + |\sigma_1|]_{32} = \sigma_2[0]_{32} = \tau$$

  And since ${ns_{32}}^{|\sigma_2|} = {ns_{32}}^{1+|\sigma_2'|} = ns_{32} \triangleright_{32} {ns_{32}}^{|\sigma_2'|}$

  $$(((\sigma_1 \circ {ns_{32}}^{|\sigma_2|}) \circ (\sigma_1 \circ \sigma_2) \circ \sigma))[|\sigma_1|]_{32} \leftarrow \tau = (\sigma_1 \circ (\tau \triangleright_{32} \epsilon) \circ {ns_{32}}^{|\sigma_2'|}) \circ (\sigma_1 \circ \sigma_2) \circ \sigma$$

  Which is what we wanted.

- If $\sigma_2 = \phi \triangleright_{64} \sigma_2'$ then the result follows by a similar argument.

■

The **stackcopy** definition is used to define a stack duplication code sequence **copyframe** $\sigma$ which emits code to duplicate $\sigma$ on the top of the stack.

$$\begin{array}{lcl}
\textbf{copyframe}\ \sigma & \stackrel{\text{def}}{=} & \texttt{salloc}\ |\sigma|; \\
& & \textbf{stackcopy}\ (\epsilon, \sigma)
\end{array}$$

| | |
|---|---|
| 64-bit values | $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{64} fv : \phi \leadsto fv'$ |
| 32-bit values | $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \tau \leadsto sv'$ |
| 64 bit operations | $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathit{dest_{64}} \leftarrow \mathit{fopr} : \phi \leadsto S$ |
| 32 bit operations | $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathit{dest_{32}} \leftarrow \mathit{opr} : \tau \leadsto S$ |
| Expressions | $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e : \tau \leadsto I; F$ |
| Heap values | $\Psi \vdash_h \mathit{hval} : \tau \leadsto I; F$ |
| Heaps | $\Psi \vdash d \leadsto H$ |
| Programs | $\vdash p : \tau \leadsto P$ |

**Figure 8.2: LIL** to **TILTAL** translation judgements

**Lemma 41 (copyframe)**
*For well-formed stack types $\sigma$ and $\sigma'$, such that $|\sigma|$ is well-defined:*

$$\Psi;\Delta;\Gamma\{\mathbf{sp}{:}\sigma \circ \sigma'\} \vdash \mathbf{copyframe}\,\sigma \Rightarrow \Gamma\{\mathbf{sp}{:}\sigma \circ \sigma \circ \sigma'\}$$

**Proof:** By construction.
By the salloc rule, it suffices to show that
$\Psi;\Delta;\Gamma\{\mathbf{sp}{:}{ns_{32}}^{|\sigma|} \circ \sigma \circ \sigma'\} \vdash \mathbf{stackcopy}(\epsilon,\sigma) \Rightarrow \Gamma\{\mathbf{sp}{:}\sigma \circ \sigma \circ \sigma'\}$
Which follows immediately by lemma 40.

∎

## 8.3 The term translation

The judgement forms used for the translation of **LIL** programs to **TILTAL** programs are listed in figure 8.3. In addition to the usual **LIL** typing contexts, all of the expression level judgements are defined with respect to an allocator $\mathcal{A}$ and two stack types $\sigma_1$ and $\sigma_2$. The allocator provides the locations of variables in registers and frame slots, and the two stack types keep track of the layout of the rest of the stack below the current frame: $\sigma_1$ describes the stack above the current handler and $\sigma_2$ describes the stack below it.

Within the bodies of functions, these stack types will generally refer to two additional free variables representing the two stack segments expected by the function (the segment above the enclosing handler, and the segment below it). By convention, I name these variable $\rho_1$ and $\rho_2$, and the stack types are expected to be well-formed in a context including these variables in addition to free type variables from the original program. So for example, an invariant of the most of the translation judgements is that $|\Delta|, \rho_1{:}ST, \rho_2{:}ST \vdash \sigma_1 : ST$, and similarly for $\sigma_2$, where $\Delta$ is the current constructor context. For brevity, I will generally abbreviate this idiom as follows:

**Definition 26**

$$\Delta^{\rho_1,\rho_2} \stackrel{\text{def}}{=} |\Delta|, \rho_1{:}ST, \rho_2{:}ST$$

The next several sections will give detailed overviews of the individual translation judgements and discuss some of the more interesting translation rules, as well as stating and proving the relevant soundness properties. The complete definition of the translation can be found at the end of the chapter.

### 8.3.1 Values

**LIL** 32 bit and 64 bit values translate into **TILTAL** 32 bit and 64 bit operands, respectively. The 32 bit value translation judgement $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \tau \rightsquigarrow sv'$ indicates that in heap context $\Psi$, constructor context $\Delta$, term context $\Gamma$, allocator $\mathcal{A}$, and stack segments $\sigma_1$ and $\sigma_2$; a **LIL** small value $sv$ of type $\tau$ translates to a **TILTAL** operand $sv'$. The 64 bit has an analogous interpretation.

#### 32 bit values

For closed values, the translation does little. For example, the translation of an integer value:

$$\frac{}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} i : \mathtt{Int} \rightsquigarrow i}$$

Similarly, for coerced values such as values injected into union types, the translation simply inductively translates the coerced value to an operand, and then applies the translated coercion.

The rules for translating variables are more interesting however, since they use the allocator to choose locations for the variable.

$$\frac{}{\Psi; \Delta; \Gamma[x{:}\tau]; \mathcal{A}[x \to \mathbf{r}], \sigma_1, \sigma_2 \vdash_{32} x : \tau \rightsquigarrow \mathbf{r}}$$

$$\frac{}{\Psi; \Delta; \Gamma[x{:}\tau]; \mathcal{A}[x \to \mathbf{sp}(i)], \sigma_2, \sigma_2 \vdash_{32} x : \tau \rightsquigarrow \mathbf{sp}(i)}$$

The translation of 32 bit **LIL** values to **TILTAL** operands is sound in the sense that given a well-behaved allocator (as defined in definition 18), a well-typed **LIL** value translates to a **TILTAL** operand that is well-typed in the translation of the term context under the allocator. More precisely:

**Theorem 18 (Soundness of the small value translation)**
*If* $\Psi; \Delta; \Gamma \vdash sv : \tau$
*and* $\mathcal{A}$ *is a good allocator for* $\Gamma$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$
*and* $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \tau \rightsquigarrow sv'$
*then*
$|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash sv' : |\tau|$

**Proof:** (theorem 18) By induction on $sv$. The proof proceeds by cases.

1. Suppose $\Psi; \Delta; \Gamma[x{:}\tau] \vdash x : \tau$ and $\Psi; \Delta; \Gamma[x{:}\tau]; \mathcal{A}[x \to \mathbf{r}], \sigma_1, \sigma_2 \vdash_{32} x : \tau \rightsquigarrow \mathbf{r}$.

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{r} : |\tau|$

   By assumption:
   $\Gamma = \Gamma_1, x{:}\tau, \Gamma_2$
   $\mathcal{A}(x) = \mathbf{r}$

So by the good allocator assumption:
$|\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}(\mathcal{A}(x)) = |\tau|$

So by the register rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{r} : |\tau|$

2. Suppose $\Psi; \Delta; \Gamma[x{:}\tau] \vdash x : \tau$ and $\Psi; \Delta; \Gamma[x{:}\tau]; \mathcal{A}[x \to \mathbf{sp}(i)], \sigma_2, \sigma_2 \vdash_{32} x : \tau \rightsquigarrow \mathbf{sp}(i)$.

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{sp}(i) : |\tau|$

   By assumption:
   $\Gamma = \Gamma_1, x{:}\tau, \Gamma_2$
   $\mathcal{A}(x) = \mathbf{sp}(i)$

   So by the good allocator assumption:
   $|\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}(\mathbf{sp}) = \sigma$
   $\sigma[i]_{32} = |\tau|$

   So by the stack slot rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{sp}(i) : |\tau|$

3. Suppose $\Psi[\ell{:}\tau]; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \ell : \tau \rightsquigarrow \ell$.

   To show:
   $|\Psi[\ell{:}\tau]|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \ell : |\tau|$

   By definition, $|\Psi[\ell{:}\tau]| = |\Psi|, \ell{:}|\tau|$ so the result follows directly by the label rule.

4. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} i : \mathtt{Int} \rightsquigarrow i$.

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash i : \mathtt{Int}$

   This follows directly by the integer rule.

5. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \mathtt{inj\_union}_c\ sv : c \rightsquigarrow \mathtt{inj\_union}_{(i,|c|)}\ sv'$.

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathtt{inj\_union}_{(i,|c|)}\ sv' : |c|$

   By inversion:
   $\Delta \vdash c \equiv \bigvee[\ldots, c_i, \ldots] : \mathrm{T}_{32}$
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : c_i \rightsquigarrow sv'$
   $\Psi; \Delta; \Gamma \vdash sv : c_i$

   By induction:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash sv' : |c_i|$

   By theorem 17 and weakening:
   $\Delta^{\rho_1,\rho_1} \vdash |c| \equiv |\bigvee[\ldots, c_i, \ldots]| : \mathrm{T}_{32}$

   By construction (injunion):
   $\Delta^{\rho_1,\rho_2} \vdash \mathtt{inj\_union}_{(i,|c|)} : |c_i| \Rightarrow |c|$

   By construction (coercion app):
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathtt{inj\_union}_{(i,|c|)}\ sv' : |c|$

6. Suppose $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} \texttt{roll}_\tau\, sv : \tau \rightsquigarrow \texttt{roll}_{|\tau|}\, sv'$.

   To show:
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash \texttt{roll}_{|\tau|}\, sv' : |\tau|$

   By inversion:
   $\Delta \vdash \tau \equiv \texttt{Rec}[\kappa](c)(c_p) : \mathrm{T}_{32}$
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : c(\texttt{Rec}[\kappa]c)c_p \rightsquigarrow sv'$
   $\Psi;\Delta;\Gamma \vdash sv : c(\texttt{Rec}[\kappa]c)c_p$

   By induction:
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash sv' : |c(\texttt{Rec}[\kappa]c)c_p|$

   By theorem 17 and weakening:
   $\Delta^{\rho_1,\rho_1} \vdash |\tau| \equiv |\texttt{Rec}[\kappa](c)(c_p)| : \mathrm{T}_{32}$

   Note that $|\texttt{Rec}[\kappa](c)(c_p)| = \texttt{Rec}[\kappa](|c|)(|c_p|)$ and $|c(\texttt{Rec}[\kappa]c)c_p| = (|c|(\texttt{Rec}[\kappa]|c|)|c_p|)$.

   By construction (roll):
   $\Delta^{\rho_1,\rho_2} \vdash \texttt{roll}_{|\tau|} : (|c|(\texttt{Rec}[\kappa]|c|)|c_p|) \Rightarrow |\tau|$

   By construction (coercion app):
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash \texttt{roll}_{|\tau|}\, sv' : |\tau|$

7. Suppose $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} \texttt{unroll}_\tau\, sv : c(\texttt{Rec}[\kappa]c)c_p \rightsquigarrow \texttt{unroll}_{|\tau|}\, sv'$.

   To show:
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash \texttt{unroll}_{|\tau|}\, sv' : |c(\texttt{Rec}[\kappa]c)c_p|$

   By inversion:
   $\Delta \vdash \tau \equiv \texttt{Rec}[\kappa](c)(c_p) : \mathrm{T}_{32}$
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \tau \rightsquigarrow sv'$
   $\Psi;\Delta;\Gamma \vdash sv : \tau$

   By induction:
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash sv' : |\tau|$

   By theorem 17 and weakening:
   $\Delta^{\rho_1,\rho_1} \vdash |\tau| \equiv |\texttt{Rec}[\kappa](c)(c_p)| : \mathrm{T}_{32}$

   Note that $|\texttt{Rec}[\kappa](c)(c_p)| = \texttt{Rec}[\kappa](|c|)(|c_p|)$ and $|c(\texttt{Rec}[\kappa]c)c_p| = (|c|(\texttt{Rec}[\kappa]|c|)|c_p|)$.

   By construction (unroll):
   $\Delta^{\rho_1,\rho_2} \vdash \texttt{unroll}_{|\tau|} : |\tau| \Rightarrow (|c|(\texttt{Rec}[\kappa]|c|)|c_p|)$

   By construction (coercion app):
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash \texttt{unroll}_{|\tau|}\, sv' : |c(\texttt{Rec}[\kappa]c)c_p|$

8. Suppose $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} \texttt{pack}\, sv\ \texttt{as}\ \tau\ \texttt{hiding}\, c : \tau \rightsquigarrow$
   $(\texttt{pack}[|\tau|]|c|)sv'$

   To show:
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash (\texttt{pack}[|\tau|]|c|)sv' : |\tau|$

   By inversion:
   $\Delta \vdash \tau \equiv \exists[\kappa](c') : \mathrm{T}_{32}$
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : c'c \rightsquigarrow sv'$
   $\Psi;\Delta;\Gamma \vdash sv : c'c$

By induction:
$|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash sv' : |c'c|$

By theorem 17 and weakening:
$\Delta^{\rho_1,\rho_1} \vdash |\tau| \equiv |\exists[\kappa](c')| : \mathrm{T}_{32}$

Note that $|\exists[\kappa]c'| = \exists[\kappa]|c'|$ and $|c\ c'| = |c|\ |c'|$.

By construction (pack):
$\Delta^{\rho_1,\rho_2} \vdash \texttt{pack}[|\tau|]|c| : |c'||c| \Rightarrow |\tau|$

By construction (coercion app):
$|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash \texttt{pack}[|\tau|]|c|sv' : |\tau|$

9. Suppose $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv[c] : c'c \leadsto sv'[|c|]$

   To show:
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash sv[c] : |c'c|$

   By inversion:
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : c'c \leadsto sv'$
   $\Psi;\Delta;\Gamma \vdash sv : \forall[\kappa](c')$
   $\Delta \vdash c : \kappa$

   By induction:
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash sv' : |\forall[\kappa](c')|$

   By theorem15 and weakening:
   $\Delta^{\rho_1,\rho_2} \vdash |c| : \kappa$

   Note that $|\forall[\kappa]c'| = \forall[\kappa]|c'|$ and $|c\ c'| = |c|\ |c'|$.

   By construction (forall instantiation):
   $|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} \vdash sv'[|c|] : |c'c|$

∎

**64 bit values**

The translation rules for 64 bit values is exactly analogous to that of 32 bit values. For variables:

$$\frac{}{\Psi;\Delta;\Gamma[x_f{:}\phi];\mathcal{A}[x_f \to \mathbf{f}],\sigma_1,\sigma_2 \vdash_{64} x_f : \phi \leadsto \mathbf{f}}$$

$$\frac{}{\Psi;\Delta;\Gamma[x_f{:}\phi];\mathcal{A}[x_f \to \mathbf{sp}(i)],\sigma_2,\sigma_2 \vdash_{32} x_f : \phi \leadsto \mathbf{sp}(i)}$$

And for constants:

$$\frac{}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_2,\sigma_2 \vdash_{32} \mathfrak{r} : \texttt{Float} \leadsto \mathfrak{r}}$$

The soundness theorem is stated and proved in much the same fashion as for 32 bit values except that no induction is required.

**Theorem 19 (Soundness of the 64 bit value translation)**

*If* $\Psi; \Delta; \Gamma \vdash fv : \phi$
*and* $\mathcal{A}$ *is a good allocator for* $\Gamma$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$
*and* $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} fv : \phi \rightsquigarrow fv'$
*then*
$|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash fv' : |\phi|$

**Proof:** By construction. The proof proceeds by cases on the last rule of the translation derivation.

1. Suppose $\Psi; \Delta; \Gamma[x_f{:}\phi]; \mathcal{A}[x_f \rightarrow \mathbf{f}], \sigma_1, \sigma_2 \vdash_{64} x_f : \phi \rightsquigarrow \mathbf{f}$

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{f} : |\phi|$

   By assumption:
   $\Gamma = \Gamma_1, x_f{:}\phi, \Gamma_2$
   $\mathcal{A}(x_f) = \mathbf{f}$

   So by the good allocator assumption:
   $|\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}(\mathcal{A}(x_f)) = |\phi|$

   So by the float register rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{f} : |\phi|$

2. Suppose $\Psi; \Delta; \Gamma[x_f{:}\phi]; \mathcal{A}[x_f \rightarrow \mathbf{sp}(i)], \sigma_2, \sigma_2 \vdash_{32} x_f : \phi \rightsquigarrow \mathbf{sp}(i)$.

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{sp}(i) : |\phi|$

   By assumption:
   $\Gamma = \Gamma_1, x{:}\tau, \Gamma_2$
   $\mathcal{A}(x) = \mathbf{sp}(i)$

   So by the good allocator assumption:
   $|\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}(\mathbf{sp}) = \sigma$
   $\sigma[i]_{64} = |\phi|$

   So by the stack slot rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathbf{sp}(i) : |\phi|$

3. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_2, \sigma_2 \vdash_{32} \mathfrak{r} : \texttt{Float} \rightsquigarrow \mathfrak{r}$.

   To show:
   $|\Psi[\ell{:}\tau]|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash \mathfrak{r} : \texttt{Float}$

   This follows immediately by construction.

■

### 8.3.2 Operations

The translation judgement for **LIL** operations takes the form $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathit{dest}_{32} \leftarrow \mathit{opr} : \tau \rightsquigarrow S$. Whereas the value translation relates **LIL** values and **TILTAL** operands, the operation translation relates a **LIL** operation $\mathit{opr}$ to both a **TILTAL** location $\mathit{dest}_{32}$ and a partial instruction sequence $S$ (as defined in section 7.2.1). The idea behind this relation is that given the machine state assumptions encoded in the allocator ($\mathcal{A}$), the partial instruction sequence $S$ implements the operation $\mathit{opr}$, leaving the result in the destination $\mathit{dest}_{32}$.

Structuring the translation in this manner is intended to improve the quality of the generated code by (among other things) eliminating un-necessary `mov` instructions. The **TILT** certifying implementation discussed in subsequent chapters uses this idea in a very similar form to that given here in the formal translation. A closely related approach to code generation is also taken by Dybvig et al. in their paper "Destination-Driven Code Generation" [DHB90].

Not all **LIL** operations are given translations by the operation translation. It simplifies the structure of the translation noticeably to translate only certain instructions (raise, handle, and the various case operations) as part of an expression. Consequently, these operations are treated as part of the expression translation.

In general, slightly different code must be produced when the destination is a stack slot versus a register. In order to keep the translation simple, I handle all stack slot destinations with a single rule.

$$\frac{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathbf{r_t} \leftarrow \mathit{opr} : \tau \rightsquigarrow S}{\begin{array}{l}\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathbf{sp}(i) \leftarrow \mathit{opr} : \tau \rightsquigarrow S; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{swrite}\ \mathbf{sp}(i), \mathbf{r_t}; \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{junk}\ \mathbf{r_t}\end{array}}$$

This rule simply translates the operation using the temporary register as the destination, then writes the value of the temporary register to the appropriate stack slot and clears the temporary register. In some cases, better code could be produced by adding additional rules for specific operation/destination pairs in which the intermediate usage of the temporary register could be eliminated.

For the inclusion of small values into the operation level, the translation produces an operand from the small value and moves it into the destination register.

$$\frac{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \mathit{sv} : \tau \rightsquigarrow \mathit{sv}'}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathbf{r} \leftarrow \mathit{sv} : \tau \rightsquigarrow \mathtt{mov}\, \mathbf{r}, \mathit{sv}'}$$

Note that in many cases, a good allocator will have assigned $\mathbf{r}$ as the location for $\mathit{sv}$ (for example, when $\mathit{sv}$ is a variable which is not live after the operation). In such cases, un-necessary move instructions will be generated. As above, additional translation rules can be added to produce better code for such cases.

A more interesting rule to consider is the translation rule for the `call` operation, which invokes a code function on a list of 64 and 32 bit arguments. Code calls are implemented by writing the code arguments onto the stack and then calling the translated code sequence. Upon return, it is necessary to move the result from the temporary register $\mathbf{r_t}$ to the destination register (since the calling convention specifies that function results are returned in $\mathbf{r_t}$). Note the use of the **movj** pseudo-instruction to ensure correctness in the case that the destination register is $\mathbf{r_t}$.

$$|\Gamma|^{\epsilon}_{\mathcal{A}} = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_e{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{sp} : \sigma_f\}$$
$$\Delta^{\rho_1,\rho_2} \vdash \sigma_f \equiv \underbrace{ns_{32} \rhd_{32} \cdots \rhd_{32} ns_{32}}_{m} \rhd_{32} \underbrace{ns_{64} \rhd_{64} \cdots ns_{64}}_{k} \rhd_{64} \sigma' : ST$$
$$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \mathtt{Code}(\tau_0, \ldots, \tau_{m-1})(\phi_0, \ldots, \phi_{k-1}) \to \tau \leadsto sv'$$
$$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_i : \tau_i \leadsto sv'_i \quad i \in 0 \ldots m-1$$
$$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} fv_i : \phi_i \leadsto fv'_i \quad i \in 0 \ldots k-1$$
$$\overline{\begin{array}{ll} \Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathbf{r} \leftarrow \mathtt{call}\ \ sv(sv_0, \ldots, sv_{m-1})(fv_0, \ldots, fv_{k-1}) : \tau \leadsto & \\ & \mathtt{fswrite}\ \mathbf{sp}(m + 2*(k-1)), fv'_{k-1} \\ & \vdots \\ & \mathtt{fswrite}\ \mathbf{sp}(m), fv'_0 \\ & \mathtt{swrite}\ \mathbf{sp}(m-1), sv'_{m-1} \\ & \vdots \\ & \mathtt{swrite}\ \mathbf{sp}(0), sv'_0 \\ & \mathbf{junkregs} \\ & \mathtt{call}\, sv'[\sigma' \circ \sigma_1, \sigma_2] \\ & \mathbf{movj}\ \mathbf{r}, \mathbf{r_t} \end{array}}$$

This rule illustrates several important ideas used in the translation. Firstly, note that the translation requires that the translation of the typing context under the allocator be a register file in which all of the registers contain only junk. This reflects the fact that all registers in this calling convention are caller-save, and hence may be overwritten during the function call. A translation using a callee-save convention could allow some or all of the registers to be occupied. The implicit effect of this rule is to force any valid translation to spill registers using the spill rule before applying a call rule (or to never map registers at all). This demonstrates a key advantage of defining the translation parametrically with respect to allocation: there is a clean separation of concerns between the expectations of the translator and the mechanism by which the register allocator satisfies those expectations.

Secondly, note that the translation expects the allocator to include space for the out-arguments in the frame. This is not required by the allocator methodology, but is convenient for the purposes of avoiding a frame pointer, as well as permitting the allocator to potentially re-use temporary slots as out-arguments. The translation expresses this requirement on the allocator by premising the translation of a `call` operation on the availability of sufficient unused slots on the top of the frame:

$$\Delta^{\rho_1,\rho_2} \vdash \sigma_f \equiv \underbrace{ns_{32} \rhd_{32} \cdots \rhd_{32} ns_{32} \rhd_{32}}_{m} \underbrace{ns_{64} \rhd_{64} \cdots ns_{64} \rhd_{64}}_{k} \sigma' : ST$$

A translation only exists if the allocator provides sufficient space in the frame.

This rule also illustrates the use of the derived **movj** partial instruction sequence from section 7.2.1. The technique used to handle stack slot destinations above implies that the translation must be prepared for the destination register to be an alias for the temporary register. This is handled here by the **movj** instruction, which performs a move on its argument registers and junks the source register only if the registers are different. Consequently, if $\mathbf{r} = \mathbf{r_t}$, then the result is left in $\mathbf{r_t}$: and if $\mathbf{r} \neq \mathbf{r_t}$, the result is moved to $\mathbf{r}$ and the $\mathbf{r_t}$ register is junked.

The rest of the operation translation rules proceed in a similar fashion. The complete translation rules can be found in section 8.4 below.

### 8.3.3 Soundness of the operation translations

The soundness theorem for the operation translation states that for a well-typed operation and a good allocator; whenever the operation is related to a destination and a partial code sequence by the translation, the partial code sequence is well-formed and leaves its result in the destination. Note that the translation assumes that the $\mathbf{r}_e$ register which is left unmapped by the allocator will contain an exception frame.

**Theorem 20 (Soundness of the operation translation.)**
*If* $\Psi;\Delta;\Gamma;\mathcal{A} \vdash \mathit{opr} : \tau\ \mathbf{opr}_{32}$ *and* $\mathcal{A}$ *is a good allocator for* $\Gamma$ *and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$ *and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$ *then:*

1. *If* $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \mathit{opr} : \tau \leadsto S$
   *then*
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau|\}$
   *where* $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\} = \Gamma_A$
2. *If* $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{sp}(i) \leftarrow \mathit{opr} : \tau \leadsto S$
   *then*
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_A\{\mathbf{sp}{:}\sigma'\}$
   *where* $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\} = \Gamma_A$ *and* $\Gamma_A(\mathbf{sp}) = \sigma$ *and* $(\sigma)[i]_{32} \leftarrow |\tau| = \sigma'$.

**Proof:** By induction on derivations. The proof proceeds by cases on the last rule used. Note that only one rule applies when the destination is a stack slot, and that when the destination is a register, at most one rule applies for each instruction form. Also note that the good allocator assumption guarantees that $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}(\mathbf{r_t}) = ns_{32}$, and hence for cases that modify $\mathbf{r_t}$, the output typing condition requires that $\mathbf{r_t}$ be coerced to $ns_{32}$.

Throughout the proof, I use $\Gamma_A$ to refer to $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$

1. Suppose $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1\sigma_2 \vdash \mathbf{sp}(i) \leftarrow \mathit{opr} : \tau \leadsto S$. The stack slot rule is the only rule that applies, so

   By inversion:
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r_t} \leftarrow \mathit{opr} : \tau \leadsto S$

   Let $\sigma = \Gamma_A(\mathbf{sp})$.

   By induction:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\tau|\}$

   By the stack write rule and lemma 34:
   $$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A\{\mathbf{r_t}{:}|\tau|\} \vdash \begin{array}{l}\texttt{swrite}\ \mathbf{sp}(i),\mathbf{r_t};\\ \textbf{junk}\ \mathbf{r_t}\end{array} \Rightarrow \Gamma_A\{\mathbf{sp}{:}(\sigma)[i]_{32} \leftarrow |\tau|\}\{\mathbf{r_t}{:}ns_{32}\}$$

   So by lemma 28 (composition):
   $$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \begin{array}{l}S;\\ \texttt{swrite}\ \mathbf{sp}(i),\mathbf{r_t};\\ \textbf{junk}\ \mathbf{r_t}\end{array} \Rightarrow \Gamma_A\{\mathbf{sp}{:}(\sigma)[i]_{32} \leftarrow |\tau|\}\{\mathbf{r_t}{:}ns_{32}\}$$

2. Suppose $opr = sv$.

   By assumption:
   $\Psi; \Delta; \Gamma; \mathcal{A} \vdash sv : \tau \ \mathbf{opr}_{32}$
   $\Delta, \Gamma, \mathcal{A} \vdash \mathbf{r} \leftarrow sv : \tau \leadsto \texttt{mov}\, \mathbf{r}, sv'$

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A} \vdash sv : \tau$
   $\Delta, \Gamma, \mathcal{A} \vdash_{32} sv : \tau \leadsto sv'$

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{mov}\, \mathbf{r}, sv' \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau|\}$

   By theorem 18:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\tau|$

   By the mov typing rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{mov}\, \mathbf{r}, sv' \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau|\}$

3. Suppose $opr = \texttt{select}\, sv$

   By assumption:
   $\Psi; \Delta; \Gamma; \mathcal{A} \vdash \texttt{select}\, sv : \tau_i \ \mathbf{opr}_{32}$
   $$\Delta, \Gamma, \mathcal{A} \vdash \mathbf{r} \leftarrow \texttt{select}\, sv : \tau_i \leadsto \begin{array}{l}\texttt{mov}\, \mathbf{r}, sv' \\ \texttt{loadr}\, \mathbf{r}, \mathbf{r}(i)\end{array}$$

   By inversion:
   $\Psi; \Delta; \Gamma \vdash sv : \times(\tau_0, \ldots, \tau_i, \ldots, \tau_n)$
   $\Delta, \Gamma, \mathcal{A} \vdash_{32} sv : \times(\tau_0, \ldots, \tau_i, \ldots, \tau_n) \leadsto sv'$

   To show:
   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \begin{array}{l}\texttt{mov}\, \mathbf{r}, sv' \\ \texttt{loadr}\, \mathbf{r}, \mathbf{r}(i)\end{array} \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau_i|\}$$

   By theorem 18:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\times(\tau_0, \ldots, \tau_i, \ldots, \tau_n)|$

   By definition:
   $|\times(\tau_0, \ldots, \tau_i, \ldots, \tau_n)| = \times(|\tau_0|, \ldots, |\tau_i|, \ldots, |\tau_n|)$

   By the mov and load typing rules:
   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \begin{array}{l}\texttt{mov}\, \mathbf{r}, sv' \\ \texttt{loadr}\, \mathbf{r}, \mathbf{r}(i)\end{array} \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau|\}$$

4. Suppose $opr = \texttt{dyntag}_c$.

   By the new tag instruction rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{dyntag}_{|c|}\ \mathbf{r} \Rightarrow \Gamma_A\{\mathbf{r}{:}|\,\texttt{Dyntag}(c)|\}$

5. Suppose $opr = \texttt{box}\, fv$.

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{64} fv : \texttt{Float} \leadsto fv'$
   $\Psi; \Delta; \Gamma \vdash fv : \phi$

To show:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{malloc}_{|\phi|}\, \mathbf{r}, fv' \Rightarrow \Gamma_A\{\mathbf{r}{:}|\,\texttt{Boxed}(\phi)|\}$

By assumption $\mathcal{A}$ is a good allocator for $\Gamma$, so

By theorem 19:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash fv' : |\phi|$

By the $\texttt{malloc}_\phi$ rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{malloc}_{|\phi|}\, \mathbf{r}, fv' \Rightarrow \Gamma_A\{\mathbf{r}{:}\,\texttt{Boxed}(|\phi|)\}$

(Note, $|\,\texttt{Boxed}(\phi)| = \texttt{Boxed}(|\phi|)$)

6. Suppose $opr = \langle sv_1, \ldots, sv_n \rangle$.

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_1 : \tau_i \rightsquigarrow sv_i'$
   $\Psi; \Delta; \Gamma \vdash sv_i : \tau_i$

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{malloc}\, \mathbf{r}, [|\tau_1|, \ldots, |\tau_n|]\langle sv_1', \ldots, sv_n' \rangle \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau_\times|\}$
   where $\tau_\times = \times[\tau_1, \ldots, \tau_n]$

   By theorem 18:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv_i : |\tau_i|$

   So the result follows directly by the malloc rule.

7. Suppose $opr = \texttt{call}\ sv(sv_0, \ldots, sv_{m-1})(fv_0, \ldots, fv_{k-1})$.

   By inversion:
   $|\Gamma|^{\epsilon}_{\mathcal{A}}(\mathbf{sp}) = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_e{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{sp} : \sigma_f\}$
   $\Delta^{\rho_1,\rho_2} \vdash \sigma_f \equiv \underbrace{ns_{32} \triangleright_{32} \cdots \triangleright_{32} ns_{32}}_{m} \triangleright_{32} \underbrace{ns_{64} \triangleright_{64} \cdots ns_{64}}_{k} \triangleright_{64} \sigma' : ST$
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \texttt{Code}(\tau_0, \ldots, \tau_{m-1})(\phi_0, \ldots, \phi_{k-1}) \to \tau \rightsquigarrow sv'$
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_i : \tau_i \rightsquigarrow sv_i' \quad i \in 0 \ldots m-1$
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} fv_i : \phi_i \rightsquigarrow fv_i' \quad i \in 0 \ldots k-1$
   $\Psi; \Delta; \Gamma \vdash sv : \texttt{Code}[\tau_0, \ldots, \tau_n][\phi_0, \ldots, \phi_k](\tau)$
   $\Psi; \Delta; \Gamma \vdash sv_i : \tau_i$
   $\Psi; \Delta; \Gamma \vdash fv_i : \phi_i$

   To show:
   $$\begin{aligned}
   |\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash\ & \texttt{fswrite}\ \mathbf{sp}(m + 2 * (k-1)), fv_{k-1}' \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau|\} \\
   & \vdots \\
   & \texttt{fswrite}\ \mathbf{sp}(m), fv_0' \\
   & \texttt{swrite}\ \mathbf{sp}(m-1), sv_{m-1}' \\
   & \vdots \\
   & \texttt{swrite}\ \mathbf{sp}(0), sv_0' \\
   & \texttt{call}\, sv'[\sigma' \circ \sigma_1, \sigma_2] \\
   & \mathbf{movj\ r}, \mathbf{r_t}
   \end{aligned}$$

   The proof proceeds by stepping through the emitted instructions and applying the appropriate typing rules. For brevity, I will simply describe the register file type after each instruction rather than restating the entire typing judgement. I also leave the inner inductions on $m$ and $k$ informal.

By theorems 19 and 18:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash fv'_i : |\phi_i|$
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_i : |\tau_i|$

By the good allocator assumption, $\Gamma_A(\mathbf{sp}) = \sigma_f \circ \sigma_1 \circ \sigma_2$.

By assumption:
$\Delta^{\rho_1,\rho_2} \vdash \sigma_f \equiv \underbrace{ns_{32} \triangleright_{32} \cdots \triangleright_{32} ns_{32}}_{m} \triangleright_{32} \underbrace{ns_{64} \triangleright_{64} \cdots ns_{64}}_{k} \triangleright_{64} \sigma' : ST \quad \text{where } \sigma_f = |\Gamma|^{\epsilon}_{\mathcal{A}}(\mathbf{sp})$

Therefore, by the `swrite` and `fswrite` rules, it suffices to show that the remainder of the instruction sequence after the stack writes is well-typed assuming that the register file has the type:

$$\Gamma_A\{\mathbf{sp}{:}\sigma_c\} \quad \text{where } \sigma_c = \tau_0 \triangleright_{32} \cdots \triangleright_{32} \tau_{m-1} \triangleright_{32} \phi_0 \triangleright_{64} \cdots \phi_{k-1} \triangleright_{64} \sigma' \circ \sigma_1 \circ \sigma_2$$

By theorem 18:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv : |\,\mathtt{Code}(\tau_0, \ldots, \tau_{m-1})(\phi_0, \ldots, \phi_{k-1}) \rightarrow \tau|$

And by definition 14:
$|\,\mathtt{Code}(\tau_0, \ldots, \tau_{m-1})(\phi_0, \ldots, \phi_{k-1}) \rightarrow \tau| =$
$\quad \forall[\rho_1{:}ST, \rho_2{:}ST].$
$\quad\quad \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\rho_2), \mathbf{r_t}{:}ns_{32},$
$\quad\quad\; \mathbf{sp}{:}\,\mathbf{cont}(|\sigma_{32} \circ \sigma_{64}|)(\rho_1)(\rho_2)(|\tau|) \triangleright_{32} (\sigma_{32} \;\circ\; (\sigma_{64} \;\circ\; \rho_1 \circ \rho_2))\} \rightarrow 0$
where $\sigma_{32} = \tau_0 \triangleright_{32} \cdots \triangleright_{32} \tau_{m-1} \triangleright_{32} \epsilon$
and $\sigma_{64} = \phi_0 \triangleright_{64} \cdots \phi_{k-1} \triangleright_{64} \epsilon$

So by the instantiation rule, $sv[\sigma' \circ \sigma_1, \sigma_2]$ has type:
$\Gamma_c \rightarrow 0 \quad$ where
$\Gamma_c = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2), \mathbf{r_t}{:}ns_{32},$
$\quad\quad \mathbf{sp}{:}\,\mathbf{cont}(|\sigma_{32} \circ \sigma_{64}|)(\sigma' \circ \sigma_1)(\sigma_2)(|\tau|) \triangleright_{32} (\sigma_{32} \;\circ\; (\sigma_{64} \;\circ\; \sigma' \circ \sigma_1 \circ \sigma_2))\}$

Let $\tau_{\mathbf{ret}} = \mathbf{cont}(|\sigma_{32} \circ \sigma_{64}|)(\sigma' \circ \sigma_1)(\sigma_2)(|\tau|)$

Note that $\Gamma_c = \Gamma_A\{\mathbf{sp}{:}\tau_{\mathbf{ret}} \triangleright_{32} \sigma_c\}$ and so the `call` rule applies. Therefore, it suffices to show that the remainder of the code sequence is well-typed under $\Gamma_{\mathbf{ret}}$, where $\tau_{\mathbf{ret}} = \Gamma_{\mathbf{ret}} \rightarrow 0$.

By lemma 38, the register file after applying the last move and junk instruction is

$$\Gamma_{\mathbf{ret}}\{\mathbf{r_t}{:}ns_{32}\}\{\mathbf{r}{:}|\tau|\}$$

But notice that by definition of **cont** (definition 13):
$\Gamma_{\mathbf{ret}} = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2), \mathbf{r_t}{:}|\tau|,$
$\quad\quad \mathbf{sp}{:}\overline{ns_{32}}^{m+2*k} \circ \sigma' \circ \sigma_1 \circ \sigma_2\}$

So $\Gamma_{\mathbf{ret}} = \Gamma_A\{\mathbf{r_t}{:}|\tau|\}$.

Finally, recall that by the good allocator assumption:
$\Gamma_A\{\mathbf{r_t}{:}ns_{32}\} = \Gamma_A$

Hence:
$\Gamma_{\mathbf{ret}}\{\mathbf{r_t}{:}ns_{32}\}\{\mathbf{r}{:}|\tau|\}$
$= \Gamma_A\{\mathbf{r_t}{:}|\tau|\}\{\mathbf{r_t}{:}ns_{32}\}\{\mathbf{r}{:}|\tau|\}$
$= \Gamma_A\{\mathbf{r_t}{:}ns_{32}\}\{\mathbf{r}{:}|\tau|\}$
$= \Gamma_A\{\mathbf{r}{:}|\tau|\}$

8. Suppose $opr = \mathtt{array}_\tau(sv_1, sv_2)$

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_1 : \mathtt{Int} \leadsto sv'_1$
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_2 : \tau \leadsto sv'_2$
   $\Psi; \Delta; \Gamma \vdash sv_1 : \mathtt{Int}$
   $\Psi; \Delta; \Gamma \vdash sv_2 : \tau$

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{malloc}_{|\tau|}\, \mathbf{r}, sv'_1, sv'_2 \Rightarrow \Gamma_A\{\mathbf{r}{:}|\mathtt{Array}_{32}(\tau)|\}$

   By theorem 18:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_1 : |\mathtt{Int}|$
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_2 : |\tau|$

   so by the malloc rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{malloc}_{|\tau|}\, \mathbf{r}, sv'_1, sv'_2 \Rightarrow \Gamma_A\{\mathbf{r}{:}\mathtt{Array}_{32}(|\tau|)\}$

9. Suppose $opr = \mathtt{farray}_\tau(sv, fv)$

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \mathtt{Int} \leadsto sv'$
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{64} fv : \phi \leadsto fv'$
   $\Psi; \Delta; \Gamma \vdash sv : \mathtt{Int}$
   $\Psi; \Delta; \Gamma \vdash fv : \phi$

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{fmalloc}_{|\phi|}\, \mathbf{r}, sv', fv' \Rightarrow \Gamma_A\{\mathbf{r}{:}|\mathtt{Array}_{64}(\tau)|\}$

   By theorems 18 and 19:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\mathtt{Int}|$
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash fv' : |\tau|$

   so by the fmalloc rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{fmalloc}_{|\phi|}\, \mathbf{r}, sv', fv' \Rightarrow \Gamma_A\{\mathbf{r}{:}|\mathtt{Array}_{64}(\tau)|\}$

10. Suppose $opr = \mathtt{sub}_\tau(sv_1, sv_2)$.

    By inversion:
    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_1 : \mathtt{Array}_{32}(\tau) \leadsto sv'_1$
    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_2 : \mathtt{Int} \leadsto sv'_2$
    $\Psi; \Delta; \Gamma \vdash sv_1 : \mathtt{Array}_{32}(\tau)$
    $\Psi; \Delta; \Gamma \vdash sv_2 : \mathtt{Int}$

    To show:
    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{sub}_{|\tau|}\, \mathbf{r}, sv'_1, sv'_2 \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau|\}$

By theorem 18:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_1 : |\texttt{Array}_{32}(\tau)|$
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_2 : \texttt{Int}$

so by the sub rule:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{sub}_{|\tau|}\, \mathbf{r}, sv'_1, sv'_2 \Rightarrow \Gamma_A\{\mathbf{r}{:}|\tau|\}$

11. Suppose $opr = \texttt{upd}_\tau(sv_1, sv_2, sv_3)$.

    By inversion:

    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_1 : \texttt{Array}_{32}(\tau) \rightsquigarrow sv'_1$
    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_2 : \texttt{Int} \rightsquigarrow sv'_2$
    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_3 : \tau \rightsquigarrow sv'_3$
    $\Psi; \Delta; \Gamma \vdash sv_1 : \texttt{Array}_{32}(\tau)$
    $\Psi; \Delta; \Gamma \vdash sv_2 : \texttt{Int}$
    $\Psi; \Delta; \Gamma \vdash sv_3 : \tau$

    To show:

    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{upd}\, sv'_1, sv'_2, sv'_3 \Rightarrow \Gamma_A\{\mathbf{r}{:}\texttt{Unit}\}$
    $\texttt{malloc}\, \mathbf{r}, [\,]\langle\rangle$

    By theorem 18:

    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_1 : |\texttt{Array}_{32}(\tau)|$
    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_2 : \texttt{Int}$
    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_3 : |\tau|$

    So by the upd rule:

    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{upd}\, sv'_1, sv'_2, sv'_3$
    $\Rightarrow \Gamma_A$

    And by the malloc rule:

    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{malloc}\, \mathbf{r}, [\,]\langle\rangle \Rightarrow \Gamma_A\{\mathbf{r}{:}\texttt{Unit}\}$

12. Suppose $opr = \texttt{fupd}_\phi(sv_1, sv_2, fv)$.

    By inversion:

    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_1 : \texttt{Array}_{32}(\tau) \rightsquigarrow sv'_1$
    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv_2 : \texttt{Int} \rightsquigarrow sv'_2$
    $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{64} fv : \phi \rightsquigarrow fv'$
    $\Psi; \Delta; \Gamma \vdash sv_1 : \texttt{Array}_{32}(\tau)$
    $\Psi; \Delta; \Gamma \vdash sv_2 : \texttt{Int}$
    $\Psi; \Delta; \Gamma \vdash fv : \phi$

    To show:

    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{fupd}\, sv'_1, sv'_2, fv' \Rightarrow \Gamma_A\{\mathbf{r}{:}\texttt{Unit}\}$
    $\texttt{malloc}\, \mathbf{r}, [\,]\langle\rangle$

    By theorems 18 and 19:

    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_1 : |\texttt{Array}_{32}(\tau)|$
    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv'_2 : \texttt{Int}$
    $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash fv' : |\phi|$

So by the fupd rule:
$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{fupd}\, sv_1', sv_2', fv'$
$\Rightarrow \Gamma_A$

And by the malloc rule:
$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{malloc}\,\mathbf{r}, []\langle\rangle \Rightarrow \Gamma_A\{\mathbf{r}{:}\texttt{Unit}\}$

■

A useful corollary states that state of the machine after executing the translation of an operation is the same as the machine state given by translating the extended context.

**Corollary 3 (Operation context extension.)**
*If* $\Psi;\Delta;\Gamma;\mathcal{A} \vdash opr : \tau\ \mathbf{opr}_{32}$
*and* $\mathcal{A}$ *is a good allocator for* $\Gamma$ *and for* $\Gamma, x{:}\tau$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$
*and* $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathcal{A}(x) \leftarrow opr : \tau \rightsquigarrow S$
*then*
$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_B$
*where* $\Gamma_A = |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$ *and* $\Gamma_B = |\Gamma, x{:}\tau|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$

**Proof:** The proof follows trivially by applying theorem 20 and observing that the good allocator assumption implies that the resulting context is equal to the translation of the extended context.

■

The proof of soundness for the 64 bit operation translation follows exactly the same form as that of the 32 bit operation translation.

**Theorem 21 (Soundness of the 64 bit operation translation.)**
*If* $\Psi;\Delta;\Gamma;\mathcal{A} \vdash fopr : \phi\ \mathbf{opr}_{64}$ *and* $\mathcal{A}$ *is a good allocator for* $\Gamma$ *and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$ *and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$ *then:*

1. *If* $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{f} \leftarrow fopr : \phi \rightsquigarrow S$
   *then*
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_A\{\mathbf{f}{:}|\phi|\}$
   *where* $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\} = \Gamma_A$
2. *If* $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{sp}(i) \leftarrow fopr : \phi \rightsquigarrow S$
   *then*
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_A\{\mathbf{sp}{:}\sigma'\}$
   *where* $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\} = \Gamma_A$ *and* $\Gamma_A(\mathbf{sp}) = \sigma$ *and* $(\sigma)[i]_{64} \leftarrow |\phi| = \sigma'$.

**Proof:** By induction on derivations. The proof proceeds by cases on the last rule used. Note that only one rule applies when the destination is a stack slot, and that when the destination is a register, at most one rule applies for each instruction form. Also note that the good allocator assumption guarantees that $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}(\mathbf{f_t}) = ns_{32}$, and hence for cases that modify $\mathbf{f_t}$, the output typing condition requires that $\mathbf{f_t}$ be coerced to $ns_{64}$.

Throughout the proof, I use $\Gamma_A$ to refer to $|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$

1. Suppose $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1\sigma_2 \vdash \mathbf{sp}(i) \leftarrow \mathit{fopr}:\phi \rightsquigarrow S$. The stack slot rule is the only rule that applies, so

   By inversion:
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{f_t} \leftarrow \mathit{fopr}:\phi \rightsquigarrow S$

   Let $\sigma = \Gamma_A(\mathbf{sp})$.

   By induction:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_A\{\mathbf{f_t}{:}|\phi|\}$

   By the stack write rule and lemma 34:
   $$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A\{\mathbf{f_t}{:}|\phi|\} \vdash \begin{array}{l}\texttt{fswrite}\ \mathbf{sp}(i),\mathbf{f_t};\\ \mathbf{junk}\ \mathbf{f_t}\end{array} \Rightarrow \Gamma_A\{\mathbf{sp}{:}(\sigma)[i]_{64} \leftarrow |\phi|\}\{\mathbf{f_t}{:}ns_{64}\}$$

   So by lemma 28 (composition):
   $$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A\{\mathbf{f_t}{:}|\phi|\} \vdash \begin{array}{l}S;\\ \texttt{fswrite}\ \mathbf{sp}(i),\mathbf{f_t};\\ \mathbf{junk}\ \mathbf{f_t}\end{array} \Rightarrow \Gamma_A\{\mathbf{sp}{:}(\sigma)[i]_{64} \leftarrow |\phi|\}\{\mathbf{f_t}{:}ns_{64}\}$$

2. Suppose $\mathit{fopr} = \mathit{fv}$.

   By assumption:
   $\Psi;\Delta;\Gamma;\mathcal{A} \vdash \mathit{fv}:\phi\ \mathbf{opr}_{64}$
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{f} \leftarrow \mathit{fv}:\tau \rightsquigarrow \texttt{fmov}\,\mathbf{f},\mathit{fv}'$

   By inversion:
   $\Psi;\Delta;\Gamma;\mathcal{A} \vdash \mathit{fv}:\phi$
   $\Psi;\Delta,\Gamma,\mathcal{A} \vdash_{64} \mathit{fv}:\phi \rightsquigarrow \mathit{fv}'$

   To show:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{fmov}\,\mathbf{f},\mathit{fv}' \Rightarrow \Gamma_A\{\mathbf{f}{:}|\phi|\}$

   By theorem 19:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \mathit{fv}':|\phi|$

   By the fmov typing rule:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{fmov}\,\mathbf{f},\mathit{sv}' \Rightarrow \Gamma_A\{\mathbf{f}{:}|\phi|\}$

3. Suppose $\mathit{fopr} = \texttt{fsub}_\phi(\mathit{sv}_1,\mathit{sv}_2)$.

   By inversion:
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} \mathit{sv}_1:\texttt{Array}_{64}(\phi) \rightsquigarrow \mathit{sv}_1'$
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} \mathit{sv}_2:\texttt{Int} \rightsquigarrow \mathit{sv}_2'$
   $\Psi;\Delta;\Gamma \vdash \mathit{sv}_1:\texttt{Array}_{64}(\phi)$
   $\Psi;\Delta;\Gamma \vdash \mathit{sv}_2:\texttt{Int}$

   To show:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{sub}_{|\phi|}\,\mathbf{f},\mathit{sv}_1',\mathit{sv}_2' \Rightarrow \Gamma_A\{\mathbf{f}{:}|\phi|\}$

   By theorem 18:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \mathit{sv}_1':|\texttt{Array}_{64}(\phi)|$
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \mathit{sv}_2':\texttt{Int}$

   so by the fsub rule:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{sub}_{|\phi|}\,\mathbf{f},\mathit{sv}_1',\mathit{sv}_2' \Rightarrow \Gamma_A\{\mathbf{f}{:}|\phi|\}$

4. Suppose $fopr = \texttt{unbox}\ sv$

   By inversion:
   $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \texttt{Boxed}(\phi) \rightsquigarrow sv'$
   $\Psi;\Delta;\Gamma \vdash sv : \texttt{Boxed}\,\phi$

   To show:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{floadr}\ \mathbf{f}, sv' \Rightarrow \Gamma_A\{\mathbf{f}{:}|\phi|\}$

   By theorem 18:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash sv' : |\texttt{Boxed}(\phi)|$

   so by the floadr rule:
   $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash \texttt{floadr}\ \mathbf{f}, sv' \Rightarrow \Gamma_A\{\mathbf{f}{:}|\phi|\}$

■

As before, a useful corollary states that state of the machine after executing the translation of a 64 bit operation is the same as the machine state given by translating the extended context.

**Corollary 4 (Operation context extension.)**
*If* $\Psi;\Delta;\Gamma;\mathcal{A} \vdash fopr : \phi\ \mathbf{opr}_{32}$
*and* $\mathcal{A}$ *is a good allocator for* $\Gamma$ *and for* $\Gamma, x_f{:}\tau$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$
*and* $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathcal{A}(x_f) \leftarrow fopr : \phi \rightsquigarrow S$
*then*
$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_A \vdash S \Rightarrow \Gamma_B$
*where* $\Gamma_A = |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$ *and* $\Gamma_B = |\Gamma, x_f{:}\phi|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$

**Proof:** The proof follows trivially by applying theorem 21 and observing that the good allocator assumption implies that the resulting context is equal to the translation of the extended context.

■

### 8.3.4 Expressions

The general expression translation judgement takes the form $\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e : \tau \rightsquigarrow I; F$. **LIL** expressions translate to **TILTAL** instruction sequences $I$, terminated either by a return, a jump, or a halt, depending on the context of occurrence of the expression. In addition, the translation of nested sub-expressions may produce a heap fragment $F$ containing additional code blocks which must be allocated in the heap at the top level. All new labels generated by the translation are assumed to be fresh.

#### Occurrence parameters

As usual, the translation relies on an allocator and two stack types describing the stack layout below the current stack frame. Additionally, the translation is parameterized by a context of occurrence $C$ indicating the context in which the translated term occurs. For the purposes of this translation, contexts range over return contexts, jump contexts, and halt contexts.

$$C ::= \texttt{ret} \mid \texttt{jmp}\, sv \mid \texttt{halt}$$

Return contexts indicate that the expression is a function body and should exit using the function return convention. Jump contexts indicate the expression occurs as a sub-block of a larger expression, and should exit by placing the return value in $\mathbf{r_t}$ and jumping to the provided address. Halt contexts indicate that the expression occurs as a program body and should exit by terminating the program.

Additional contexts are possible: for example the implementation uses a special context to efficiently translate boolean expressions whose only use is to decide branches. The implementation also uses this mechanism to implement proper tail-recursion: function calls occurring at the end of expressions translated in the return context are tail calls. The formal translation does not address this optimization since it adds nothing interesting to the development. As with the use of the destination parameter in the operation translation, this idea is closely related to the techniques used by Dybvig, et al [DHB90].

Most of the translation inference rules are parametric with respect to the context of occurrence. The exceptions are the rules for the small value base case of expressions, where the the appropriate terminator for the instruction sequence is chosen.

### Return values

For return contexts, the translation inference rule requires that the top element of the stack (below the frame) be an appropriate continuation type[2]. The code sequence emitted moves the return value to $\mathbf{r_t}$, de-allocates the stack frame, coerces the registers and in-arguments to nonsense, and returns.

$$\frac{\begin{array}{c}\Delta^{\rho_1,\rho_2} \vdash \sigma_1 \equiv (\mathbf{cont}(\overline{m})(\sigma_1')(\sigma_2)(|\tau_r|)) \triangleright_{32} \sigma_1' : ST \\ \Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \tau_r \rightsquigarrow sv' \quad \mathrm{frmsz}(\mathcal{A}) = n\end{array}}{\begin{array}{ll}\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{ret}} sv : \tau_r \rightsquigarrow & \texttt{mov}\ \mathbf{r_t}, sv'; \\ & \texttt{sfree}\ n; \\ & \mathbf{junkregs}; \\ & \mathbf{junkstack}\ 1 \ldots m; \\ & \texttt{ret};\end{array}}$$

For expressions translated in a jump context, the context of occurrence carries with it the destination of the jump. The translation rule makes no explicit assumptions about the state of the stack, but insists that the type of the destination be appropriate for a jump. The translated code moves the result value to the temporary register and jumps to the destination. Note that the destination is assumed to have been previously instantiated with all of the necessary type variables: it is not sound to simply instantiate it with the current typing context $\Delta$, since $\Delta$ may contain more type variables than the destination is expecting.

$$\frac{\begin{array}{c}|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash sv_l : \Gamma\{\mathbf{r_t}{:}|\tau|\} \to 0 \\ \Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \tau \rightsquigarrow sv'\end{array}}{\begin{array}{ll}\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{jmp}\, sv_l} sv : \tau \rightsquigarrow & \texttt{mov}\ \mathbf{r_t}, sv'; \\ & \texttt{jmp}\ sv_l;\end{array}}$$

[2]It is an invariant of the translation as stated that this should always be true. Note that the soundness of the translation does not rely on this invariant, since the existence of a translation derivation is predicated on it being satisfied. However, a proof of completeness of the term translation would require that this assumption be shown valid.

Finally, expressions translated in a halt context move the result value to the temporary register, de-allocate the stack frame, and halt the program. Note that the translation insists that the stack segments below the current frame be empty.

$$\frac{\Psi; \Delta; \Gamma; \mathcal{A}, \epsilon, \epsilon \vdash_{32} sv : \tau_r \rightsquigarrow sv' \quad \mathrm{frmsz}(\mathcal{A}) = n}{\Psi; \Delta; \Gamma; \mathcal{A}, \epsilon, \epsilon \vdash_{\mathtt{halt}} sv : \tau_r \rightsquigarrow \begin{array}{l}\mathtt{mov}\ \mathbf{r_t}, sv';\\ \mathtt{sfree}\ n;\\ \mathtt{halt}_{\tau_r};\end{array}}$$

### Spilling

For the most part, all of the exception rules are syntax directed. However, I include one non-syntax directed rule in the translation to give the allocator more flexibility in its choices of locations. The non-deterministic *spill* rule allows a shift to a new good allocator so long as a code sequence can be found that re-arranges the frame appropriately.

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma; \mathcal{A}', \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow I\\ |\Psi|; \Delta^{\rho_1,\rho_1}; |\Gamma|^{\sigma_1 \circ \sigma_2}_{\mathcal{A}} \vdash S \Rightarrow |\Gamma|^{\sigma_1 \circ \sigma_2}_{\mathcal{A}'}\\ \text{where } \mathcal{A}' \text{ is a good allocator for e}\end{array}}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow S; I}$$

This rule is intended to capture the fact that a register allocator may wish to insert code into the instruction stream at various points (such as spill and restore code). Since various instructions (such as function calls) insist that the register file be empty, this rule gives the allocator an opportunity to obey this constraint without forcing all variables live across the call to be permanently stack resident.

### Operations

The translation of ordinary **LIL** operation binding demonstrates the interaction between the allocator and the destination based translation used for operations.

$$\frac{\begin{array}{c}\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathcal{A}(x) \leftarrow opr : \tau_i \rightsquigarrow S\\ \Psi; \Delta; \Gamma, x{:}\tau_i; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow I\end{array}}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \mathtt{let}_\tau\, x = opr\ \mathtt{in}\, e : \tau \rightsquigarrow \begin{array}{l}S;\\ I\end{array}}$$

The operation *opr* is translated with respect to the destination chosen by the allocator for $x$ to produce a partial instruction sequence $S$ that implements the operation and leaves the result in the chosen destination. The rest of the expression is translated to produce an instruction sequence $I$, to which $S$ is prepended. Uses of $x$ within $e$ will be translated via the allocator to references to the chosen location for $x$.

### Operations translated as part of the expression translation

Some operations are not given direct translations as part of the operation translation, but are instead translated directly by the expression translation. This is done to keep the operation trans-

$$
\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv:\texttt{Dyn}\rightsquigarrow sv' \quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_1:\texttt{Dyntag}(\tau_1)\rightsquigarrow sv' \\ \Psi;\Delta;\Gamma,x_1{:}\times[\texttt{Dyntag}(\tau_1),\tau_1];\mathcal{A},\sigma_1,\sigma_2\vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_1:\tau_x\rightsquigarrow I_1 \\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_2:\tau_x\rightsquigarrow I_2 \\ \Psi;\Delta;\Gamma,x{:}\tau;\mathcal{A},\sigma_1,\sigma_2\vdash_C e:\tau\rightsquigarrow I \\ |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}=\Gamma_A\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_C \texttt{let}\,x=\texttt{dyncase}(sv)(sv_1\Rightarrow x_1.e_1,\_\Rightarrow e_2)\,\texttt{in}\,e:\tau\rightsquigarrow \\ \qquad\qquad\qquad \texttt{mov}\,\mathbf{r_t},sv' \\ \qquad\qquad\qquad \texttt{brdyn}\ \mathbf{r_t},sv_1',\ell_1[(\Pi_1\Delta),\sigma_1,\sigma_2] \\ \qquad\qquad\qquad \mathbf{junk}\ \mathbf{r_t} \\ \qquad\qquad\qquad I_2 \\ \qquad\qquad\quad \ell_1{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}\times[\texttt{Dyntag}(|\tau_1|),|\tau_1|]] \\ \qquad\qquad\qquad \mathbf{srmov}\,\mathcal{A}(x_1),\mathbf{r_t} \\ \qquad\qquad\qquad \mathbf{junk}\ \mathbf{r_t} \\ \qquad\qquad\qquad I_1 \\ \qquad\qquad\quad \ell_{\mathbf{end}}{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_x|] \\ \qquad\qquad\qquad \mathbf{srmov}\ \mathcal{A}(x),\mathbf{r_t} \\ \qquad\qquad\qquad \mathbf{junk}\ \mathbf{r_t} \\ \qquad\qquad\qquad I\end{array}}
$$

**Figure 8.3:** Exception case translation in **TILTAL**.

lation simpler, usually because the operations generate additional heap allocated code blocks. Sum and exception case analysis and handlers all fall into this category.

The translation of exception case analysis (figure 8.3) produces two new code blocks and a code sequence. The code sequence compares the tag of an exception in register $\mathbf{r_t}$ to a known tag, and if equal branches to the first additional code block: $\ell_1$. This code block moves the refined exception into the destination assigned to $x_1$ by the allocator, and then executes the translation of the body of the case statement $e_1$. Notice though that the context parameter for the expression translation is changed to indicate the block should exit by jumping to the merge point $\ell_{\mathbf{end}}$. Since code blocks must be closed, all of the free type variables must be threaded through the blocks: hence the instantiation of the destination label $\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]$. (I use the notation $\Pi_1\Delta$ to indicate the constructor variables bound in the domain of $\Delta$.)

In the case that the branch is not taken, the other branch of the case is executed. Note that it too is translated in a context with the merge point as its jump destination. The remainder of the expression, $e$, is translated in the original context of occurrence, and then is emitted with the merge point label $\ell_{\mathbf{end}}$, with the free type variables suitably abstracted. Note that by convention the continuation expects its argument in $\mathbf{r_t}$.

The translation of sum cases follows essentially the same form, but with substantially more cases. The principle additional complication is that numerous branches must be generated to distinguish between the various tags and tagged record within the union type. No effort is made in the formal translation to emit these in an especially efficient fashion (for example by using general comparisons and intermediate branches instead of simple equality tests).

Exception handling is handled in a relatively inefficient manner for simplicity in the translation (see figure 8.4). The translation insists that the allocator spill all registers to the stack at every handler site, and then pushes the old exception handler frame onto the stack and copies the entire frame over it. The handled expression is translated with a jump continuation that forwards control to a postlude which cleans up the stack and restores the old handler before jumping to the merge point. The handler code itself does a similar cleanup before executing the handler code, which is translated with the final merge point as its continuation.

Raising exceptions is also dealt with at the expression level, not because it requires additional code blocks to be emitted, but rather because exception raising code ends with an unconditional jump and hence does not fit into the syntactic category of partial instruction sequences. To raise an exception, the exception packet must be placed in $\mathbf{r}_1$, the handler stack must be restored from the exception frame before jumping to the handler, also obtained from the frame.

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \mathtt{Dyn} \leadsto sv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\, x = \mathtt{raise}_\tau\, sv\, \mathtt{in}\, e : \tau \leadsto \begin{array}{l}\mathtt{mov}\,\mathbf{r}_1, sv\\ \mathtt{loadr}\,\mathbf{r_t}, \mathbf{r}_e(1)\\ \mathtt{mov}\,\mathbf{sp}, \mathbf{r_t}\\ \mathtt{loadr}\,\mathbf{r_t}, \mathbf{r}_e(0)\\ \mathtt{jmp}\,\mathbf{r_t}\end{array}}$$

#### Type operations

The remainder of the expression translation inference rules deal with the type instructions, such as unpacking existentials and refining types using the LX based primitives. Since these constructs are essentially the same in **LIL** and **TILTAL**, the translation does little interesting work. For example, the refinement operation for pairs simply translates the rest of the expression and prepends a pair refinement instruction.

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \alpha : \kappa_1 \times \kappa_2\\ \left.\begin{array}{l}\Psi;\Delta,\beta{:}\kappa_1,\gamma{:}\kappa_2;\\ \Gamma[\langle\beta,\gamma\rangle/\alpha];\\ \mathcal{A},\sigma_1[\langle\beta,\gamma\rangle/\alpha],\sigma_2[\langle\beta,\gamma\rangle/\alpha]\end{array}\right\} \vdash_{C[\langle\beta,\gamma\rangle/\alpha]} e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \leadsto I\end{array}}{\Psi;\Delta,\alpha{:}\kappa_1\times\kappa_2;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\langle\beta,\gamma\rangle = c\, \mathtt{in}\, e : \tau \leadsto \begin{array}{l}\mathtt{refine}\langle\beta,\gamma\rangle = |c|;\\ I\end{array}}$$

The rest of the type instruction rules are similarly un-interesting and are not described here further. A complete listing of all of the expression translation is given in section 8.4.

### 8.3.5 Soundness of the expression translation

The soundness theorem for the expression translation says that if a well-formed **LIL** expression translates to an instruction sequence and a heap fragment, then both the instruction sequence and the heap fragment are well-formed. Note that the heap context $\Psi$ is extended with the additional bindings introduced by the new heap when typechecking the instruction sequence. As before, I assume that the allocator is a good allocator for the expression, and that the stack segment parameters are well-formed.

$$
\frac{
\begin{gathered}
\Psi;\Delta;\Gamma;\mathcal{A},\epsilon,\sigma_3 \vdash_{\mathtt{jmp}\,\ell_{\mathbf{post}}[(\Pi_1\Delta),\epsilon,\sigma_h]} e_1 : \tau_x \leadsto I_1; F_1 \\
\Psi;\Delta;\Gamma,x_2{:}\,\mathtt{Dyn};\mathcal{A},\sigma_1,\sigma_2 \vdash_{\mathtt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_f\circ\sigma_1,\sigma_2]} e_2 : \tau_x \leadsto I_2; F_2 \\
\Psi;\Delta;\Gamma,x{:}\tau_x;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e : \tau \leadsto I; F \\
\Gamma_A = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2), \mathbf{sp}{:}\sigma_f\circ\sigma_1\circ\sigma_2, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}\} \\
\text{where} \quad \Gamma_A = |\Gamma|^{\sigma_1\circ\sigma_2}_{\mathcal{A}}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2) \\
\text{and} \quad \Gamma_B = |\Gamma|^{\sigma_3}_{\mathcal{A}}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_3) \\
\text{and} \quad \sigma_f = |\Gamma|^{\epsilon}_{\mathcal{A}}(\mathbf{sp}) \\
\text{and} \quad \sigma_3 = \mathbf{Exnptr}(\sigma_2) \triangleright_{32} \sigma_f\circ\sigma_1\circ\sigma_2 \\
\text{and} \quad \sigma_h = \sigma_f\circ\sigma_3
\end{gathered}
}{
\begin{array}{l}
\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\,x = \mathtt{handle}_{\tau_x}(e_1, x_2.e_2)\,\mathtt{in}\,e : \tau \leadsto \\
\qquad \mathtt{push}\ \mathbf{r}_e \\
\qquad \mathbf{stackcopy}\ (\mathbf{Exnptr}(\sigma_2)) \triangleright_{32} \sigma_f \\
\qquad \mathtt{sfree}\ 1 \\
\qquad \mathtt{malloc}\ \mathbf{r}_e[\mathbf{Exnhndler}(\sigma_h),\sigma_h]\langle \ell_{\mathbf{handle}}[\Pi_1\Delta,\rho_1,\rho_2],\mathbf{sp}\rangle \\
\qquad I_1 \\
\quad \ell_{\mathbf{handle}}{:}\forall[\Delta,\rho_1,\rho_2].\{\mathbf{r}_1{:}\,\mathtt{Dyn}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{r}_e{:}ns_{32}, \mathbf{sp}{:}\sigma_h, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}\} \\
\qquad \mathbf{srmov}\ \mathcal{A}(x_2), \mathbf{r}_1 \\
\qquad \mathtt{sfree}(\mathrm{frmsz}(\mathcal{A})) \\
\qquad \mathtt{pop}\ \mathbf{r}_e \\
\qquad \mathbf{junk}\ \mathbf{r}_1 \\
\qquad I_2 \\
\quad \ell_{\mathbf{post}}{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_B[\mathbf{r_t}{:}|\tau_x|] \\
\qquad \mathtt{sfree}(\mathrm{frmsz}(\mathcal{A})) \\
\qquad \mathtt{pop}\ \mathbf{r}_e \\
\qquad \mathtt{jmp}\ \ell_{\mathbf{end}}[\Pi_1\Delta,\sigma_1,\sigma_2] \\
\quad \ell_{\mathbf{end}}{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_x|] \\
\qquad \mathbf{srmov}\ \mathcal{A}(x), \mathbf{r_t} \\
\qquad \mathbf{junk}\ \mathbf{r_t} \\
\qquad I; \\
\quad F; \\
\quad F_1; \\
\quad F_2
\end{array}
}
$$

**Figure 8.4:** Exception handler implementation in **TILTAL**.

**Theorem 22 (Soundness of the expression translation.)**

*If* $\mathcal{A}$ *is a good allocator for* $e$
*and* $\Psi; \Delta; \Gamma \vdash e : \tau$ **exp**
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
*and* $\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$
*and* $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow I; F$
*then*
$|\Psi| \vdash F : \Psi_F$
*and* $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash I$ **ok**
*where* $\Gamma_A = |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$

**Proof:** By induction on derivations. The proof proceeds by cases on the last rule of the translation derivation. I generally identify the rule under consideration by listing the conclusion of the rule, except where ambiguous.

Throughout the proof, let $\Gamma_A = |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$.

1. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\mathtt{ret}} sv : \tau_r \rightsquigarrow \mathtt{mov}\ \mathbf{r_t}, sv';$
   $\mathtt{sfree}\ n;$
   **junkregs**;
   **junkstack** $1 \ldots m$;
   $\mathtt{ret}$

   To show:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{mov}\ \mathbf{r_t}, sv';$ **ok**
   $\mathtt{sfree}\ n;$
   **junkregs**;
   **junkstack** $1 \ldots m$;
   $\mathtt{ret}$

   By assumption:
   $\Psi; \Delta; \Gamma; \mathcal{A} \vdash sv : \tau_r$
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \tau_r \rightsquigarrow sv'$
   $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 \equiv (\mathbf{cont}(\overline{m})(\sigma_1')(\sigma_2)(|\tau_r|)) \triangleright_{32} \tau_1 \triangleright_{32} \ldots \triangleright_{32} \tau_m \triangleright_{32} \sigma_1' : ST$

   So by theorem 18:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\tau_r|$

   By the $\mathtt{mov}$ typing rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{mov}\ \mathbf{r_t}, sv' \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}$

   By the good allocator assumption, $\Gamma_A(\mathbf{sp}) = \sigma_f \circ (\sigma_1 \circ \sigma_2)$ and $\mathrm{frmsz}(()\mathcal{A}) = |\sigma_f|$

   So by the $\mathtt{sfree}$ typing rule:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\} \vdash \mathtt{sfree}\ n \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}\{\mathbf{sp}{:}(\sigma_1 \circ \sigma_2)\}$

   By lemma 34:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}\{\mathbf{sp}{:}(\sigma_1 \circ \sigma_2)\} \vdash \mathbf{junkregs} \Rightarrow$
   $\Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}\{\mathbf{sp}{:}(\sigma_1 \circ \sigma_2)\}\{\mathbf{r}_1{:}ns_{32}\}\{\mathbf{r}_2{:}ns_{32}\}\{\mathbf{f}_1{:}ns_{64}\}\{\mathbf{f}_2{:}ns_{64}\}$

   Let $\Gamma_A' = \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}\{\mathbf{sp}{:}(\sigma_1 \circ \sigma_2)\}\{\mathbf{r}_1{:}ns_{32}\}\{\mathbf{r}_2{:}ns_{32}\}\{\mathbf{f}_1{:}ns_{64}\}\{\mathbf{f}_2{:}ns_{64}\}$

   By assumption:
   $\Delta^{\rho_1,\rho_2} \vdash \sigma_1 \equiv (\mathbf{cont}(\overline{m})(\sigma_1')(\sigma_2)(|\tau_r|)) \triangleright_{32} \tau_1 \triangleright_{32} \ldots \triangleright_{32} \tau_m \triangleright_{32} \sigma_1' : ST$

So by lemma 35:

$$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma'_A \vdash \mathbf{junkstack}\ 1 \ldots m \Rightarrow$$
$$\Gamma'_A\{\mathbf{sp}{:}(ns_{32}{}^{\overline{m}}) \circ \sigma'_1 \circ \sigma_2)\}$$

And by definition:

$$\mathbf{cont}(\overline{m})(\sigma'_1)(\sigma_2)(|\tau_r|) = \Gamma_{\mathbf{ret}} \rightarrow 0$$

where

$$\Gamma_{\mathbf{ret}} = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2), \mathbf{r_t}{:}|\tau_r|, \mathbf{sp}{:}ns_{32}{}^{\overline{m}} \circ \sigma'_1 \circ \sigma_2\}$$

So by the return rule, it suffices to show that :

$$\Gamma_r\{\mathbf{sp}{:}ns_{32}{}^{\overline{m}} \circ \sigma'_1 \circ \sigma_2\} = \Gamma_{\mathbf{ret}}$$

By the good allocator assumption:

$$\Gamma_r(\mathbf{r}_e) = \Gamma_A(\mathbf{r}_e) = \mathbf{Exnptr}(\sigma_2)$$

So by definition:

$$\Gamma_r = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2), \mathbf{r_t}{:}|\tau_r|, \mathbf{sp}{:}ns_{32}{}^{\overline{m}} \circ \sigma'_1 \circ \sigma_2\} = \Gamma_{\mathbf{ret}}$$

2. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\mathtt{jmp}\, sv_l} sv : \tau \rightsquigarrow \begin{array}{l}\mathtt{mov}\ \ \mathbf{r_t}, sv' \\ \mathtt{jmp}\ \ sv_l\end{array}$

   To show:

   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \begin{array}{l}\mathtt{mov}\ \ \mathbf{r_t}, sv'\ \mathbf{ok} \\ \mathtt{jmp}\ \ sv_l\end{array}$$

   By assumption:

   $$\Psi; \Delta; \Gamma; \mathcal{A} \vdash sv : \tau$$
   $$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \tau \rightsquigarrow sv'$$
   $$|\Psi|; \Delta^{\rho_1,\rho_2}; |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2} \vdash sv_l : \Gamma\{\mathbf{r_t}{:}|\tau|\} \rightarrow 0$$

   So by theorem 18:

   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\tau|$$

   By the `mov` typing rule:

   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{mov}\ \ \mathbf{r_t}, sv' \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}$$

   And by the `jmp` typing rule:

   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\} \vdash \mathtt{jmp}\ \ sv'_l\ \mathbf{ok}$$

3. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \epsilon, \epsilon \vdash_{\mathtt{halt}} sv : \tau_r \rightsquigarrow \begin{array}{l}\mathtt{mov}\ \ \mathbf{r_t}, sv' \\ \mathtt{sfree}\ \ n \\ \mathtt{halt}_{\tau_r}\end{array}$

   By inversion:

   $$\Psi; \Delta; \Gamma; \mathcal{A}, \epsilon, \epsilon \vdash_{32} sv : \tau_r \rightsquigarrow sv'$$
   $$\mathrm{frmsz}(\mathcal{A}) = n$$
   $$\Psi; \Delta; \Gamma; \mathcal{A} \vdash sv : \tau_r$$

   By theorem 18:

   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\tau_r|$$

   So by the `mov` typing rule:

   $$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{mov}\ \ \mathbf{r_t}, sv' \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}$$

By the good allocator assumption:
$\Gamma_A(\mathbf{sp}) = |\Gamma|_{\mathcal{A}}^{\epsilon\circ\epsilon} = \sigma_f \circ \epsilon$
where $|\sigma_f| = \text{frmsz}(()\mathcal{A}) = n$

So by the `sfree` typing rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_r| \vdash \texttt{sfree}\ n \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}\{\mathbf{sp}{:}\epsilon\}$

And by the `halt` typing rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_r|\}\{\mathbf{sp}{:}\epsilon\} \vdash \texttt{halt}_{|\tau_r|}\ \mathbf{ok}$

4. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \texttt{let}_\tau\, x = \mathit{opr}\,\texttt{in}\,e : \tau \leadsto (S; I); F$.

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathcal{A}(x) \leftarrow \mathit{opr} : \tau_i \leadsto S$
   $\Psi; \Delta; \Gamma, x{:}\tau_i; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \leadsto I \Psi; \Delta; \Gamma \vdash \mathit{opr} : \tau_i\ \mathbf{opr}_{32}$
   $\Psi; \Delta; \Gamma, x{:}\tau_i \vdash e : \tau\ \mathbf{exp}$

   And by the good allocator assumption, $\mathcal{A}$ is a good allocator for $\Gamma$.

   So by corollary 3
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash S \Rightarrow \Gamma_B$

   Where $\Gamma_B = |\Gamma, x{:}\tau_i|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$

   By the definition of a good allocator for an expression, $\mathcal{A}$ is a good allocator for the sub-expression $e$, so:

   By induction:
   $|\Psi| \vdash F : \Psi_F$
   $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_B \vdash I\ \mathbf{ok}$

   And by lemma 29 (partial instruction sequence completion) and heap context weakening:
   $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash (S; I)\ \mathbf{ok}$

5. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \texttt{let}_\tau\, x_f = \mathit{fopr}\,\texttt{in}\,e : \tau \leadsto (S; I); F$.

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash \mathcal{A}(x_f) \leftarrow \mathit{fopr} : \phi \leadsto S$
   $\Psi; \Delta; \Gamma, x_f{:}\phi; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \leadsto I \Psi; \Delta; \Gamma \vdash \mathit{fopr} : \phi\ \mathbf{opr}_{32}$
   $\Psi; \Delta; \Gamma, x_f{:}\tau \vdash e : \tau\ \mathbf{exp}$

   And by the good allocator assumption, $\mathcal{A}$ is a good allocator for $\Gamma$.

   So by corollary 4
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash S \Rightarrow \Gamma_B$

   Where $\Gamma_B = |\Gamma, x_f{:}\phi|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$

   By the definition of a good allocator for an expression, $\mathcal{A}$ is a good allocator for the sub-expression $e$, so:

   By induction:
   $|\Psi| \vdash F : \Psi_F$
   $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_B \vdash I\ \mathbf{ok}$

   And by lemma 29 (partial instruction sequence completion) and heap context weakening:
   $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash (S; I)\ \mathbf{ok}$

6. Suppose $\Psi; \Delta; \Gamma; \mathcal{A} \vdash_C \texttt{let}[\alpha, x] = \texttt{unpack}\, sv \;\texttt{in}\, e : \tau \rightsquigarrow \texttt{unpack}\ [\alpha, \mathbf{r_t}]sv';$

$$\begin{array}{l} \mathbf{srmov}\ \ \mathcal{A}(x), \mathbf{r_t}; \\ \mathbf{junk}\ \ \mathbf{r_t}; \\ I; \\ F \end{array}$$

   By inversion:
   $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \exists[\kappa][c] \rightsquigarrow sv'$
   $\Delta, \alpha{:}\kappa; \Gamma, x{:}(c\alpha); \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow I; F$
   $\Psi; \Delta; \Gamma \vdash sv : \exists[\kappa](c)$
   $\Psi; \Delta, \alpha{:}\kappa; \Gamma, x{:}(c\alpha) \vdash e : \tau'\ \mathbf{exp} \quad \alpha \notin fv(\tau')$

   By theorem 18:
   $|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\exists[\kappa](c)|$

   Since $|\exists[\kappa](c)| = \exists[\kappa](|c|)$, by the unpack rule it suffices to show:
   $|\Psi|; \Delta, \alpha{:}\kappa^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}c\ \alpha\} \vdash \mathbf{srmov}\ \ \mathcal{A}(x), \mathbf{r_t};\ \mathbf{ok}$

$$\begin{array}{l} \mathbf{junk}\ \ \mathbf{r_t}; \\ I \end{array}$$

   By lemma 36, the machine state after the **srmov** instruction is (informally written):
   $\Gamma_A\{\mathbf{r_t}{:}|c\ \alpha|\}\{\mathcal{A}(x){:}|c\ \alpha|\}$

   And by lemma 34, the machine state after the **junk** instruction is (also informally written):
   $\Gamma_A\{\mathbf{r_t}{:}|c\ \alpha|\}\{\mathcal{A}(x){:}|c\ \alpha|\}\{\mathbf{r_t}{:}ns_{32}\} = \Gamma_A\{\mathcal{A}(x){:}|c\ \alpha|\}$
   (since $\Gamma_A(\mathbf{r_t}) = ns_{32}$).

   But by the good allocator assumption:
   $\Gamma_A\{\mathcal{A}(x){:}|c\ \alpha|\} = \Gamma_B$
   where $\Gamma_B = |\Gamma, x{:}c\ \alpha|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_1)\}$

   Finally, by induction and heap context weakening:
   $|\Psi| \vdash F : \Psi_F$
   $|\Psi|, \Psi_F; \Delta, \alpha{:}\kappa^{\rho_1,\rho_2}; \Gamma_B \vdash I\ \mathbf{ok}$

7. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \texttt{let}\langle\beta, \gamma\rangle = c\,\texttt{in}\,e : \tau \rightsquigarrow I; F$ and $\Delta \vdash c \equiv \langle c_1, c_2\rangle : \kappa_1 \times \kappa_2$.

   By inversion:
   $\Psi; \Delta, ; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e[c_1, c_2/\beta, \gamma] : \tau \rightsquigarrow I; F$
   $\Psi; \Delta; \Gamma \vdash e[c_1, c_2/\beta, \gamma] : \tau\ \mathbf{exp}$

   By induction:
   $|\Psi| \vdash F : \Psi_F$
   $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash I\ \mathbf{ok}$

8. Suppose

$$\Psi; \Delta, \alpha{:}\kappa_1 \times \kappa_2; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \texttt{let}\langle\beta, \gamma\rangle = c\,\texttt{in}\,e : \tau \rightsquigarrow \begin{array}{l} \texttt{refine}\langle\beta, \gamma\rangle = |c|; \\ I; \\ F \end{array}$$

   and $\Delta, \alpha{:}\kappa_1 \times \kappa_2 \vdash c \equiv \alpha : \kappa_1 \times \kappa_2$.

By inversion:

$$\left.\begin{array}{l}\Psi; \Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2;\\ \Gamma[\langle\beta,\gamma\rangle/\alpha];\\ \mathcal{A}, \sigma_1[\langle\beta,\gamma\rangle/\alpha], \sigma_2[\langle\beta,\gamma\rangle/\alpha]\end{array}\right\} \vdash_{C[\langle\beta,\gamma\rangle/\alpha]} e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \rightsquigarrow I; F$$

$\Psi; \Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2, \Delta'; \Gamma[\langle\beta,\gamma\rangle/\alpha] \vdash e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha]$ **exp**

By assumption:
$\Delta, \alpha{:}\kappa_1 \times \kappa_2^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
$\Delta, \alpha{:}\kappa_1 \times \kappa_2^{\rho_1,\rho_2} \vdash \sigma_2 : ST$

So by substitution :
$\Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2^{\rho_1,\rho_2} \vdash \sigma_1[|\langle\beta,\gamma\rangle|/\alpha] : ST$
$\Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2^{\rho_1,\rho_2} \vdash \sigma_2[|\langle\beta,\gamma\rangle|/\alpha] : ST$

So by induction :
$|\Psi| \vdash F : \Psi_F$
$|\Psi|, \Psi_F; \Delta, \beta{:}\kappa_1, \gamma{:}\kappa_2^{\rho_1,\rho_2}; \Gamma_S \vdash I$ **ok**
where $\Gamma_S = |\Gamma[\langle\beta,\gamma\rangle/\alpha]|_{\mathcal{A}}^{(\sigma_1\circ\sigma_2)[|\langle\beta,\gamma\rangle|/\alpha]}$

By lemma 33:
$\Gamma_S = |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}[|\langle\beta,\gamma\rangle|/\alpha] = \Gamma_A[|\langle\beta,\gamma\rangle|/\alpha]$

So by the pair refinement rule:
$|\Psi|, \Psi_F; \Delta, \alpha{:}\kappa_1 \times \kappa_2^{\rho_1,\rho_2}; \Gamma_A \vdash \begin{array}[t]{l}\mathtt{refine}\langle\beta,\gamma\rangle = |c|;\\ I\end{array}$ **ok**

9. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \mathtt{let\,fold}\,\beta = c\,\mathtt{in}\,e : \tau \rightsquigarrow I; F$ and $\Delta \vdash c \equiv \mathtt{fold}_{\mu j.\kappa}\, c' : \mu j.\kappa$.

   By inversion:
   $\Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_1 \vdash_C e[c'/\beta] : \tau \rightsquigarrow I; F$
   $\Psi; \Delta; \Gamma \vdash e[c'/\beta] : \tau$ **exp**

   By induction:
   $|\Psi| \vdash F : \Psi_F$
   $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash I$ **ok**

10. Suppose

$$\Psi; \Delta, \alpha{:}\mu j.\kappa, \Delta'; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \mathtt{let\,fold}\,\beta = c\,\mathtt{in}\,e : \tau \rightsquigarrow \begin{array}[t]{l}\mathtt{refine}[\mathtt{fold}\,\beta]c;\\ I;\\ F\end{array}$$

   and $\Delta, \alpha{:}\mu j.\kappa, \Delta' \vdash c \equiv \alpha : \mu j.\kappa$.

   By inversion:

$$\left.\begin{array}{l}\Psi; \Delta, \beta{:}\kappa[\mu j.\kappa/j], \Delta';\\ \Gamma[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha];\\ \mathcal{A},\\ \sigma_1[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha],\\ \sigma_1[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha]\end{array}\right\} \vdash_{C[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha]} e[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] : \tau[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] \rightsquigarrow I; F$$

   $\Psi; \Delta, \beta{:}\kappa[\mu j.\kappa/j], \Delta'; \Gamma[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] \vdash e[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] : \tau[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha]$ **exp**

By assumption:
$\Delta, \alpha{:}\mu j.\kappa, \Delta'^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
$\Delta, \alpha{:}\mu j.\kappa, \Delta'^{\rho_1,\rho_2} \vdash \sigma_2 : ST$

So by substitution :
$\Delta, \beta{:}\kappa[\mu j.\kappa/j], \Delta'^{\rho_1,\rho_2} \vdash \sigma_1[|\,\mathtt{fold}_{\mu j.\kappa}\,\beta|/\alpha] : ST$
$\Delta, \beta{:}\kappa[\mu j.\kappa/j], \Delta'^{\rho_1,\rho_2} \vdash \sigma_2[|\,\mathtt{fold}_{\mu j.\kappa}\,\beta|/\alpha] : ST$

So by induction :
$|\Psi| \vdash F : \Psi_F$
$|\Psi|, \Psi_F; \Delta, \beta{:}\kappa[\mu j.\kappa/j], \Delta'^{\rho_1,\rho_2}; \Gamma_S \vdash I$ **ok**
where $\Gamma_S = |\Gamma[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha]|_{\mathcal{A}}^{(\sigma_1\circ\sigma_2)[|\,\mathtt{fold}_{\mu j.\kappa}\,\beta|/\alpha]}$

By lemma 33:
$\Gamma_S = |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}[|\,\mathtt{fold}_{\mu j.\kappa}\,\beta|/\alpha] = \Gamma_A[|\,\mathtt{fold}_{\mu j.\kappa}\,\beta|/\alpha]$

So by the fold refinement rule:
$|\Psi|, \Psi_F; \Delta, \alpha{:}\mu j.\kappa, \Delta'^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{refine}[\mathtt{fold}\,\beta]c;$ **ok**
$I$

11. Suppose $\Psi; \Delta; \Gamma \vdash_C \mathtt{let}_\tau\, \mathtt{inj}_i\,\beta = (c, sv)\,\mathtt{in}\, e : \tau \rightsquigarrow I; F$ and $\Delta \vdash c \equiv \mathtt{inj}_i^{+[\kappa_1,\ldots,\kappa_i,\ldots,\kappa_n]}\, c' : +[\kappa_1, \ldots, \kappa_i, \ldots, \kappa_n]$.

By inversion:
$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e[c'/\beta] : \tau \rightsquigarrow I; F$
$\Psi; \Delta; \Gamma \vdash e[c'/\beta] : \tau$ **exp**

By induction:
$|\Psi| \vdash F : \Psi_F$
$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash I$ **ok**

12. Suppose

$$\Psi; \Delta, \alpha{:}+[\kappa_1, \ldots, \kappa_n], \Delta' \vdash_C \mathtt{let}_\tau \mathtt{inj}_i \beta = (c, sv)\,\mathtt{in}\, e : \tau \rightsquigarrow \begin{array}{l}\mathtt{refine}[\mathtt{inj}_i\,\beta]|c|, sv';\\ I;\\ F\end{array}$$

and $\Delta, \alpha{:}\kappa_1 + \kappa_2, \Delta' \vdash c \equiv \alpha : \kappa_1 + \kappa_2$.

By inversion:
$$\left.\begin{array}{c}\Psi; \Delta, \beta{:}\kappa_i, \Delta';\\ \Gamma[\mathtt{inj}_i\,\beta/\alpha];\\ \mathcal{A}, \sigma_1[\mathtt{inj}_i\,\beta/\alpha], \sigma_2[\mathtt{inj}_i\,\beta/\alpha]\end{array}\right\} \vdash_{C[\mathtt{inj}_i\,\beta/\alpha]} e[\mathtt{inj}_i\,\beta/\alpha] : \tau[\mathtt{inj}_i\,\beta/\alpha] \rightsquigarrow I; F$$
$\Psi; \Delta, \beta{:}\kappa_j, \Delta'; \Gamma[\mathtt{inj}_j\,\beta/\alpha]; \mathcal{A}, \sigma_1[\mathtt{inj}_j\,\beta/\alpha], \sigma_2[\mathtt{inj}_j\,\beta/\alpha] \vdash_{32} sv[\mathtt{inj}_j\,\beta/\alpha] : \mathtt{Void} \rightsquigarrow sv'$
$j \in 1 \ldots i-1, i+1 \ldots n$
$\Psi; \Delta, \beta{:}\kappa_i, \Delta'; \Gamma[\mathtt{inj}_i\,\beta/\alpha] \vdash e[\mathtt{inj}_i\,\beta/\alpha] : \tau[\mathtt{inj}_i\,\beta/\alpha]$ **exp**
$\Psi; \Delta, \beta{:}\kappa_j, \Delta'; \Gamma[\mathtt{inj}_j\,\beta/\alpha] \vdash sv[\mathtt{inj}_j\,\beta/\alpha] : \mathtt{Void}$
$j \in 1 \ldots i-1, i+1 \ldots n$

By assumption:
$\Delta, \alpha{:}+[\kappa_1, \ldots, \kappa_n], \Delta'^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
$\Delta, \alpha{:}+[\kappa_1, \ldots, \kappa_n], \Delta'^{\rho_1,\rho_2} \vdash \sigma_2 : ST$

So by substitution :
$\Delta, \beta{:}\kappa_i, \Delta'^{\rho_1,\rho_2} \vdash \sigma_1[|\,\mathtt{inj}_i\,\beta|/\alpha] : ST$
$\Delta, \beta{:}\kappa_i, \Delta'^{\rho_1,\rho_2} \vdash \sigma_2[|\,\mathtt{inj}_i\,\beta|/\alpha] : ST$
$j \in 1 \ldots n$

So by induction :
$|\Psi| \vdash F : \Psi_F$
$|\Psi|, \Psi_F; \Delta, \beta{:}\kappa_i, \Delta'^{\rho_1,\rho_2}; \Gamma_S \vdash I$ **ok**
where $\Gamma_S = |\Gamma[\mathtt{inj}_i\,\beta/\alpha]|_{\mathcal{A}}^{(\sigma_1\circ\sigma_2)[|\,\mathtt{inj}_i\,\beta|/\alpha]}$

By lemma 33:
$\Gamma_S = |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}[|\,\mathtt{inj}_i\,\beta|/\alpha] = \Gamma_A[|\,\mathtt{inj}_i\,\beta|/\alpha]$

By theorem 18 (small value translation):
$|\Psi|; \Delta\beta{:}\kappa_i, \Delta'^{\rho_1,\rho_2}; \Gamma_V \vdash sv' : \mathtt{Void}$
$j \in 1 \ldots i-1, i+1 \ldots n$
where $\Gamma_S = |\Gamma[\mathtt{inj}_i\,\beta/\alpha]|_{\mathcal{A}}^{(\sigma_1\circ\sigma_2)[|\,\mathtt{inj}_i\,\beta|/\alpha]}$

So by the sum refinement rule:
$|\Psi|, \Psi_F; \Delta, \alpha{:} + [\kappa_1, \ldots, \kappa_n], \Delta'^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{refine}[\mathtt{inj}_i\,\beta]|c|, sv';$ **ok**
$I$

13. Suppose the last rule used was the `case` translation rule. The translation of a case statement produces a number of new blocks and an instruction sequence. By lemma 30 it suffices to show that each of these is well-formed when the heap-context is extended with the types of the new blocks. Note that I assume all labels are fresh. I leave informal the inner induction on the number of branches $n$.

Let
$$\begin{aligned}\Psi_F = \Psi(&F; F_1; \ldots; F_n),\\ &\ell_0{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_0|],\\ &\ell_1{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_1|],\\ &\vdots\\ &\ell_n{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_n|],\\ &\ell_{\mathbf{end}}{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau|]\end{aligned}$$

(a) Heap fragments.

To show:
$$\begin{aligned}|\Psi| \vdash\ &F; \ : \Psi(F; F_1; \ldots; F_n)\\ &F_1;\\ &\vdots\\ &F_n\end{aligned}$$

By inversion:
$\Psi; \Delta; \Gamma, x_i{:}\tau_i; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\mathtt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_i : \tau \rightsquigarrow I_i; F_i \quad i \in 1 \ldots n$
$\Psi; \Delta; \Gamma, x_i{:}\tau_i \vdash e_i : \tau$ **exp**
$\Psi; \Delta; \Gamma, x{:}\tau; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau_e \rightsquigarrow I; F$
$\Psi; \Delta; \Gamma, x{:}\tau \vdash e : \tau_e$ **exp**

By induction:

$|\Psi| \vdash F_i : \Psi_i$

$|\Psi| \vdash F : \Psi'$

So by lemma 30:

$|\Psi| \vdash F; F_1; \ldots; F_n : \Psi', \Psi_1, \ldots, \Psi_n$

(b) The prelude code:

To show:

$$
\begin{array}{lll}
|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash & \texttt{mov}\,\mathbf{r_t}, sv' & \mathbf{ok} \\
& \texttt{brtag}_0\,\mathbf{r_t}, \ell_0[(\Pi_1\Delta), \sigma_1, \sigma_2] & \\
& \vdots & \\
& \texttt{brtag}_{k-1}\,\mathbf{r_t}, \ell_{k-1}[(\Pi_1\Delta), \sigma_1, \sigma_2] & \\
& \texttt{brtgd}_k\,\mathbf{r_t}, \ell_k[(\Pi_1\Delta), \sigma_1, \sigma_2] & \\
& \vdots & \\
& \texttt{brtag}_{n-1}\,\mathbf{r_t}, \ell_{n-1}[(\Pi_1\Delta), \sigma_1, \sigma_2] & \\
& \texttt{mov}\,\mathbf{r_t}, \texttt{forgetunion}\,\mathbf{r_t} & \\
& \texttt{jmp}\,\ell_n[(\Pi_1\Delta), \sigma_1, \sigma_2] &
\end{array}
$$

By inversion:

$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \bigvee[\tau_1, \ldots, \tau_n] \rightsquigarrow sv'$

$\Delta \vdash \tau_i \equiv \texttt{Tag}(i) : \mathrm{T}_{32} \quad i \in 1 \ldots (j-1)$

$\Delta \vdash \tau_i \equiv \times[\texttt{Tag}(i), \tau_i'] : \mathrm{T}_{32} \quad i \in j \ldots n$

$\Psi; \Delta; \Gamma \vdash sv : \bigvee[\tau_0, \ldots, \tau_k]$

By theorem 18:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\bigvee[\tau_0, \ldots, \tau_k]|$

So by the mov rule:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{mov}\ \mathbf{r_t}, sv' \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\bigvee[\tau_0, \ldots, \tau_k]|\}$

By theorem 17 and weakening:

$\Delta^{\rho_1,\rho_2} \vdash |\tau_i| \equiv \texttt{Tag}(i) : \mathrm{T}_{32} \quad i \in 1 \ldots (j-1)$

$\Delta^{\rho_1,\rho_2} \vdash |\tau_i| \equiv \times[\texttt{Tag}(i), \tau_i'] : \mathrm{T}_{32} \quad i \in j \ldots n$

By assumption, $\ell_i{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_i|]$,

So the `brtag` and `brtgd` instructions are well-formed and have well-formed targets.

After $n-1$ tag comparisons, $\mathbf{r_t}{:}\bigvee[|\tau_n|]$. (There is an inner induction here about which I am being informal).

Therefore, by the `forgetunion` and the `mov` rules:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}\bigvee[|\tau_n|]\} \vdash \texttt{mov}\ \mathbf{r_t}, \texttt{forgetunion}\,\mathbf{r_t} \Rightarrow \Gamma_A\{\mathbf{r_t}{:}||\tau_n||\}$

So the final jump is well-formed as well.

(c) $\ell_i$, for $i \in 0 \ldots n$.

To show:

$$
\begin{array}{ll}
|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A[\mathbf{r_t}{:}|\tau_i|] \vdash & \mathbf{srmov}\,\mathcal{A}(x_i), \mathbf{r_t}\ \mathbf{ok} \\
& \mathbf{junk}\ \mathbf{r_t} \\
& I_i
\end{array}
$$

By inversion:

$\Psi; \Delta; \Gamma, x_i{:}\tau_i; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_i : \tau \rightsquigarrow I_i; F_i \quad i \in 1 \ldots n$

$\Psi; \Delta; \Gamma, x_i{:}\tau_i \vdash e_i : \tau\ \mathbf{exp}$

By induction:
$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_i \vdash I_i$ **ok** $\quad i \in 1 \dots n$
where $\Gamma_i = |\Gamma, x_i{:}\tau_i|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}$

By the good allocator assumption:
$\Gamma_i(\mathbf{r_t}) = ns_{32}$ and
$\Gamma_i = |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathcal{A}(x_i){:}|\tau_i|\} = \Gamma_A\{\mathcal{A}(x_i){:}|\tau_i|\}$

So by lemmas 36 and 34:
$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_i|\} \vdash \begin{array}{l}\mathbf{srmov}\, \mathcal{A}(x_i), \mathbf{r_t}\ \mathbf{ok} \quad i \in 1 \dots n\\ \mathbf{junk}\ \mathbf{r_t}\\ I_i\end{array}$$

(d) $\ell_{\mathbf{end}}$

To show:
$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A[\mathbf{r_t}{:}|\tau|] \vdash \begin{array}{l}\mathbf{srmov}\ \ \mathcal{A}(x), \mathbf{r_t}\ \mathbf{ok}\\ \mathbf{junk}\ \mathbf{r_t}\\ I\end{array}$$

By inversion:
$\Psi; \Delta; \Gamma, x{:}\tau; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau_e \rightsquigarrow I; F$
$\Psi; \Delta; \Gamma, x{:}\tau \vdash e : \tau_e$ **exp**

By induction:
$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_x \vdash$
**ok**where $\Gamma_x = |\Gamma, x{:}\tau|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}$

By the good allocator assumption:
$\Gamma_x(\mathbf{r_t}) = ns_{32}$ and
$\Gamma_x = |\Gamma|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathcal{A}(x){:}|\tau|\} = \Gamma_A\{\mathcal{A}(x){:}|\tau|\}$

So by lemmas 36 and 34:
$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau|\} \vdash \begin{array}{l}\mathbf{srmov}\, \mathcal{A}(x), \mathbf{r_t}\ \mathbf{ok}\\ \mathbf{junk}\ \mathbf{r_t}\\ I\end{array}$$

14. Suppose the last rule used was the `dyncase` translation rule. The translation of an exception case statement produces a number of new blocks and an instruction sequence. By lemma 30 it suffices to show that each of these is well-formed when the heap-context is extended with the types of the new blocks. Note that I assume all labels are fresh.

Let
$$\begin{array}{l}\Psi_F = \Psi(F; F_1; F_n),\\ \quad \ell_1{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:} \times [\mathtt{Dyntag}(|\tau_1|), |\tau_1|]]\\ \quad \ell_{\mathbf{end}}{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_x|]\end{array}$$

(a) Heap fragments.

To show:
$$|\Psi| \vdash \begin{array}{l}F;\\ F_1;\\ F_2\end{array} : \Psi(F; F_1; F_2)$$

By inversion:

$\Psi; \Delta; \Gamma, x_1{:} \times [\texttt{Dyntag}(\tau_1), \tau_1]; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_1 : \tau_x \rightsquigarrow I_1; F_1$

$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_2 : \tau_x \rightsquigarrow I_2; F_2$

$\Psi; \Delta; \Gamma, x{:}\tau_x; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow I; F$

$\Psi; \Delta; \Gamma, x_1{:} \times [\texttt{Dyntag}(\tau_1), \tau_1] \vdash e_1 : \tau_x\ \mathbf{exp}$

$\Psi; \Delta; \Gamma \vdash e_2 : \tau_x\ \mathbf{exp}$

$\Psi; \Delta; \Gamma, x{:}\tau_x \vdash e : \tau\ \mathbf{exp}$

By induction:

$|\Psi| \vdash F_1 : \Psi_1$

$|\Psi| \vdash F_2 : \Psi_2$

$|\Psi| \vdash F : \Psi'$

So by lemma 30:

$|\Psi| \vdash F; F_1; F_2 : \Psi', \Psi_1, \Psi_2$

(b) The "otherwise" case:

To show:

$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \begin{array}{l} \texttt{mov}\,\mathbf{r_t}, sv' \\ \texttt{brdyn}\ \mathbf{r_t}, sv_1', \ell_1[(\Pi_1\Delta), \sigma_1, \sigma_2] \\ \mathbf{junk}\ \mathbf{r_t} \\ I_2 \end{array} \mathbf{ok}$$

By inversion:

$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \texttt{Dyn} \rightsquigarrow sv'$

$\Psi; \Delta; \Gamma \vdash sv : \texttt{Dyn}$

By theorem 18:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\,\texttt{Dyn}\,|$

So by the mov rule:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{mov}\ \mathbf{r_t}, sv' \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\,\texttt{Dyn}\,|\}$

By assumption:

$\ell_1{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A\{\mathbf{r_t}{:} \times [\texttt{Dyntag}(|\tau_1|), |\tau_1|]\}$

So by the `brdyn` rule:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \texttt{brdyn}\ \mathbf{r_t}, sv_1', \ell_1[(\Pi_1\Delta), \sigma_1, \sigma_2] \Rightarrow \Gamma_A\{\mathbf{r_t}{:}||\,\texttt{Dyn}\,||\}$

And by lemma 34:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathbf{junk}\ \mathbf{r_t} \Rightarrow \Gamma_A\{\mathbf{r_t}{:}||\,\texttt{Dyn}\,||\}\{\mathbf{r_t}{:}ns_{32}\}$

Note that $\Gamma_A\{\mathbf{r_t}{:}||\,\texttt{Dyn}\,||\}\{\mathbf{r_t}{:}ns_{32}\} = \Gamma_A$.

By induction:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash I_2\ \mathbf{ok}$

So the result follows directly by the instruction sequencing rules.

(c) The "equal" case – $\ell_1$:

To show:

$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:} \times [\texttt{Dyntag}(|\tau_1|), |\tau_1|]\} \vdash \begin{array}{l} \mathbf{srmov}\,\mathcal{A}(x_1), \mathbf{r_t}\ \mathbf{ok} \\ \mathbf{junk}\ \mathbf{r_t} \\ I_1 \end{array}$$

By inversion:

$\Psi; \Delta; \Gamma, x_1{:} \times [\texttt{Dyntag}(\tau_1), \tau_1]; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_1 : \tau_x \rightsquigarrow I_1; F_1$

$\Psi; \Delta; \Gamma, x_1{:} \times [\texttt{Dyntag}(\tau_1), \tau_1] \vdash e_1 : \tau_x\ \mathbf{exp}$

By induction:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_1 \vdash I_1$ **ok**where $\Gamma_1 = |\Gamma, x_1{:}\tau_1|^{\sigma_1 \circ \sigma_2}_{\mathcal{A}}$

By the good allocator assumption:

$\Gamma_1(\mathbf{r_t}) = ns_{32}$ and

$\Gamma_1 = |\Gamma|^{\sigma_1\circ\sigma_2}_{\mathcal{A}}\{\mathcal{A}(x_1){:}| \times [\mathtt{Dyntag}(\tau_1), \tau_1]|\} = \Gamma_A\{\mathcal{A}(x_1){:} \times [\mathtt{Dyntag}(|\tau_1|), |\tau_1|]\}$

So it suffices to show that:

$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:} \times [\mathtt{Dyntag}(|\tau_1|), |\tau_1|]\} \vdash \begin{array}{l}\mathbf{srmov}\, \mathcal{A}(x_i), \mathbf{r_t} \Rightarrow \Gamma_B \\ \mathbf{junk}\ \mathbf{r_t}\end{array}$$

where $\Gamma_B = \Gamma_A\{\mathcal{A}(x_1){:} \times [\mathtt{Dyntag}(|\tau_1|), |\tau_1|]\}$

Which follows by lemmas 36 and 34.

(d) The continuation – $\ell_{\mathbf{end}}$:

To show:

$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A[\mathbf{r_t}{:}|\tau_x|] \vdash \begin{array}{l}\mathbf{srmov}\ \ \mathcal{A}(x), \mathbf{r_t}\ \mathbf{ok} \\ \mathbf{junk}\ \ \mathbf{r_t} \\ I\end{array}$$

By inversion:

$\Psi; \Delta; \Gamma, x{:}\tau_x; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow I; F$

$\Psi; \Delta; \Gamma, x{:}\tau_x \vdash e : \tau$ **exp**

By induction:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_x \vdash$

**ok**where $\Gamma_x = |\Gamma, x{:}\tau_x|^{\sigma_1\circ\sigma_2}_{\mathcal{A}}$

By the good allocator assumption:

$\Gamma_x(\mathbf{r_t}) = ns_{32}$ and

$\Gamma_x = |\Gamma|^{\sigma_1\circ\sigma_2}_{\mathcal{A}}\{\mathcal{A}(x){:}|\tau_x|\} = \Gamma_A\{\mathcal{A}(x){:}|\tau_x|\}$

So by lemmas 36 and 34:

$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_x|\} \vdash \begin{array}{l}\mathbf{srmov}\, \mathcal{A}(x), \mathbf{r_t}\ \mathbf{ok} \\ \mathbf{junk}\ \ \mathbf{r_t} \\ I\end{array}$$

15. Suppose

$$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C \mathtt{let}\, x = \underset{\tau}{\mathtt{raise}}\, sv\, \mathtt{in}\, e : \tau \rightsquigarrow \begin{array}{l}\mathtt{mov}\, \mathbf{r}_1, sv \\ \mathtt{loadr}\, \mathbf{r_t}, \mathbf{r}_e(1) \\ \mathtt{mov}\, \mathbf{sp}, \mathbf{r_t} \\ \mathtt{loadr}\, \mathbf{r_t}, \mathbf{r}_e(0) \\ \mathtt{jmp}\, \mathbf{r_t}\end{array}$$

By inversion:

$\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \mathtt{Dyn} \rightsquigarrow sv'$

$\Psi; \Delta; \Gamma \vdash sv : \mathtt{Dyn}$

By theorem 18:

$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash sv' : |\,\mathtt{Dyn}\,|$

By assumption:

$\Gamma_A(\mathbf{sp}) = \sigma_f \circ \sigma_1 \circ \sigma_2$

$\Gamma_A(\mathbf{r}_e) = \mathbf{Exnptr}(\sigma_2)$

By definition:
$\mathbf{Exnptr}(\sigma_2) = \times[\mathbf{Exnhndler}(\sigma_2), \sigma_2]$
$\mathbf{Exnhndler}(\sigma_2) = \forall[].\{\mathbf{r}_1\colon \mathtt{Dyn}, \mathbf{sp}{:}\sigma_2\} \rightarrow 0$

By the mov rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \mathtt{mov}\ \ \mathbf{r}_1, sv' \Rightarrow \Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,|\}$

By the load rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,| \vdash \mathtt{loadr}\, \mathbf{r_t}, \mathbf{r}_e(1) \Rightarrow \Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,|\}\{\mathbf{r_t}{:}\sigma_2\}$

By the stack load rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,|\}\{\mathbf{r_t}{:}\sigma_2\} \vdash \mathtt{mov}\, \mathbf{sp}, \mathbf{r_t} \Rightarrow \Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,|\}\{\mathbf{r_t}{:}\sigma_2\}\{\mathbf{sp}{:}\sigma_2\}$

By the load rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,|\}\{\mathbf{r_t}{:}\sigma_2\}\{\mathbf{sp}{:}\sigma_2\} \vdash \mathtt{loadr}\, \mathbf{r_t}, \mathbf{r}_e(0) \Rightarrow$
$\Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,|\}\{\mathbf{sp}{:}\sigma_2\}\{\mathbf{r_t}\colon \mathbf{Exnhndlr}(\sigma_2)\}$

By the jump rule:
$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r}_1{:}|\,\mathtt{Dyn}\,|\}\{\mathbf{sp}{:}\sigma_2\}\{\mathbf{r_t}\colon \mathbf{Exnhndlr}(\sigma_2)\} \vdash \mathtt{jmp}\, \mathbf{r_t}\ \mathbf{ok}$

16. Suppose the last rule used was the `handle` translation rule. The translation of an exception handler produces a number of new blocks and an instruction sequence. By lemma 30 it suffices to show that each of these is well-formed when the heap-context is extended with the types of the new blocks. Note that I assume all labels are fresh.
Let
$\Gamma_B = |\Gamma|^{\sigma_3}_{\mathcal{A}}\{\mathbf{r}_e\colon \mathbf{Exnptr}(\sigma_3)$
$\sigma_f = |\Gamma|^{\epsilon}_{\mathcal{A}}(\mathbf{sp})$
$\sigma_3 = \mathbf{Exnptr}(\sigma_2) \triangleright_{32} \sigma_f \circ \sigma_1 \circ \sigma_2$
$\sigma_h = \sigma_f \circ \sigma_3$
$\Psi_F = \Psi(F; F_1; F_n),$
$\ell_{\mathbf{handle}}{:}\forall[\Delta, \rho_1, \rho_2].\{\mathbf{r}_1\colon \mathtt{Dyn}, \mathbf{sp}{:}\sigma_h, \ldots\}$
$\ell_{\mathbf{post}}{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_B[\mathbf{r_t}{:}|\tau_x|]$
$\ell_{\mathbf{end}}{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_x|]$

(a) Heap fragments.
To show:
$|\Psi| \vdash F;\ : \Psi(F; F_1; F_2)$
$F_1;$
$F_2$
By inversion:
$\Psi; \Delta; \Gamma; \mathcal{A}, \epsilon, \sigma_3 \vdash_{\mathtt{jmp}\,\ell_{\mathbf{post}}[(\Pi_1\Delta),\epsilon,\sigma_h]} e_1 : \tau_x \leadsto I_1; F_1$
$\Psi; \Delta; \Gamma, x_2\colon \mathtt{Dyn}; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\mathtt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_f\circ\sigma_1,\sigma_2]} e_2 : \tau_x \leadsto I_2; F_2$
$\Psi; \Delta; \Gamma, x{:}\tau_x; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \leadsto I; F$
$\Psi; \Delta; \Gamma \vdash e_1 : \tau_x\ \mathbf{exp}$
$\Psi; \Delta; \Gamma, x_2\colon \mathtt{Dyn} \vdash e_2 : \tau_x\ \mathbf{exp}$
$\Psi; \Delta; \Gamma, x{:}\tau_x \vdash e : \tau\ \mathbf{exp}$
By induction:
$|\Psi| \vdash F_1 : \Psi_1$
$|\Psi| \vdash F_2 : \Psi_2$
$|\Psi| \vdash F : \Psi'$

So by lemma 30:
$|\Psi| \vdash F; F_1; F_2 : \Psi', \Psi_1, \Psi_2$

(b) The handled code block:

To show:

$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash \begin{array}{l} \texttt{push}\ \mathbf{r}_e \\ \textbf{stackcopy}\ (\textbf{Exnptr}(\sigma_2)) \triangleright_{32} \sigma_f \\ \texttt{sfree}\ 1 \\ \texttt{malloc}\ \mathbf{r}_e[\textbf{Exnhndler}(\sigma_h), \sigma_h]\langle \ell_{\textbf{handle}}[\Pi_1\Delta, \rho_1, \rho_2], \mathbf{sp}\rangle \\ I_1 \end{array} \quad \textbf{ok}$$

By assumption:
$\Gamma_A(\mathbf{sp}) = \sigma_f \circ \sigma_1 \circ \sigma_2$
$\Gamma_A(\mathbf{r}_e) = \textbf{Exnptr}(\sigma_2)$

The introductory sequence of instructions only modifies the stack and the exception register, so for brevity I leave the rest of the typing context implicit and only describe the types of the stack register and the exception register after the instructions.

| Instruction | $\mathbf{sp}$ type | $\mathbf{r}_e$ type | Comment |
|---|---|---|---|
| `push` | $\sigma_3 = (\textbf{Exnptr}(\sigma_2)) \triangleright_{32} \sigma_f \circ \sigma_1 \circ \sigma_2$ | $\textbf{Exnptr}(\sigma_2)$ | By the `push` rule |
| **stackcopy** | $(\textbf{Exnptr}(\sigma_2)) \triangleright_{32} \sigma_f \circ \sigma_3$ | $\textbf{Exnptr}(\sigma_2)$ | By lemma 41 |
| `sfree` | $\sigma_h = \sigma_f \circ \sigma_3$ | $\textbf{Exnptr}(\sigma_2)$ | By the `sfree` rule |
| `malloc` | $\sigma_h$ | $\textbf{Exnptr}(\sigma_h)$ | By the `malloc` rule |

So it suffices to show that:
$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{sp}{:}\sigma_h\}\{\mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_3)\} \vdash I_1\ \textbf{ok}$

But note that :
$|\Gamma|_{\mathcal{A}}^{\sigma_h} = |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{sp}{:}\sigma_g\} = \Gamma_A\{\mathbf{sp}{:}\sigma_g\}$

So the desired result follows by induction.

(c) The handler block:

Let $\Gamma_H = \{\mathbf{r}_1{:}\,\texttt{Dyn}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{r}_e{:}ns_{32}, \mathbf{sp}{:}\sigma_h, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}\}$

To show:

$$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_H \vdash \begin{array}{l} \textbf{srmov}\ \ \mathcal{A}(x_2), \mathbf{r}_1\ \textbf{ok} \\ \texttt{sfree}(\text{frmsz}(\mathcal{A})) \\ \texttt{pop}\ \mathbf{r}_e \\ \textbf{junk}\ \mathbf{r}_1 \\ I_2 \end{array}$$

| Instr | machine state | Comment |
|---|---|---|
| **srmov** | $\Gamma_H\{\mathcal{A}(x_2){:}\,\texttt{Dyn}\}$ | By lemma 36 |
| `sfree` | $\Gamma_H\{\mathcal{A}(x_2){:}\,\texttt{Dyn}\}\{\mathbf{sp}{:}\sigma_3\}$ | By the `sfree` rule |
| **pop** | $\Gamma_H\{\mathcal{A}(x_2){:}\,\texttt{Dyn}\}\{\mathbf{sp}{:}\sigma_f \circ \sigma_1 \circ \sigma_2\}\{\mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2)\}$ | By the `pop` rule |
| **junk** | $\ldots\{\mathbf{r}_1{:}ns_{32}\}$ | By lemma 34 |

But note that by assumption:
$\Gamma_A$
$= |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2)\}$
$= \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2), \mathbf{sp}{:}\sigma_f \circ \sigma_1 \circ \sigma_2, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}\}$

And by definition:

$\Gamma_H\{\mathbf{sp}{:}\sigma_f \circ \sigma_1 \circ \sigma_2\}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}\{\mathbf{r}_1{:}ns_{32}\}$
$= \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2), \mathbf{sp}{:}\sigma_f \circ \sigma_1 \circ \sigma_2, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}\}$
$= \Gamma_A$

So by the good allocator assumption:

$\Gamma_H\{\mathcal{A}(x_2){:}\,\mathtt{Dyn}\}\{\mathbf{sp}{:}\sigma_f \circ \sigma_1 \circ \sigma_2\}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$
$= \Gamma_A\{\mathcal{A}(x_2){:}\,\mathtt{Dyn}\}$
$= |\Gamma, x_2{:}\,\mathtt{Dyn}\,|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$

Finally, by induction:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathcal{A}(x_2){:}\,\mathtt{Dyn}\} \vdash I_2\ \mathbf{ok}$

(d) The postlude block:

To show:

$$
\begin{array}{ll}
|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_B[\mathbf{r_t}{:}|\tau|] \vdash & \mathtt{sfree}(\mathrm{frmsz}(\mathcal{A})) \qquad \mathbf{ok} \\
& \mathtt{pop}\ \mathbf{r}_e \\
& \mathtt{jmp}\ \ell_{\mathbf{end}}[\Pi_1\Delta, \sigma_1, \sigma_2]
\end{array}
$$

Note that $\Gamma_B(\mathbf{sp}) = \sigma_h = \sigma_f \circ \sigma_3$.

So by the `sfree` rule:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_B\{\mathbf{r_t}{:}|\tau|\} \vdash \mathtt{sfree}(\mathrm{frmsz}(\mathcal{A})) \Rightarrow \Gamma_B\{\mathbf{r_t}{:}|\tau|\}\{\mathbf{sp}{:}\sigma_3\}$

By the `pop` rule:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_B\{\mathbf{r_t}{:}|\tau|\}\{\mathbf{sp}{:}\sigma_3\} \vdash \mathtt{pop}\ \mathbf{r}_e \Rightarrow$
$\Gamma_B\{\mathbf{r_t}{:}|\tau|\}\{\mathbf{sp}{:}\sigma_f \circ \sigma_1 \circ \sigma_2\}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$

Note that:

$\Gamma_B\{\mathbf{r_t}{:}|\tau|\}\{\mathbf{sp}{:}\sigma_f \circ \sigma_1 \circ \sigma_2\}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$
$= |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r_t}{:}|\tau|\}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$
$= \Gamma_A\{\mathbf{r_t}{:}|\tau|\}$

And by assumption:

$\ell_{\mathbf{end}}{:}\forall[|\Delta|, \rho_1{:}ST, \rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_x|]$

So by the `jmp` rule:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau|\} \vdash \mathtt{jmp}\ \ell_{\mathbf{end}}[\Pi_1\Delta, \sigma_1, \sigma_2]\ \mathbf{ok}$

(e) The continuation block:

To show:

$$
\begin{array}{ll}
|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_x|\} \vdash & \mathbf{srmov}\ \ \mathcal{A}(x), \mathbf{r_t}\ \mathbf{ok} \\
& \mathbf{junk}\ \ \mathbf{r_t} \\
& I
\end{array}
$$

By lemma 36:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_x|\} \vdash \mathbf{srmov}\ \ \mathcal{A}(x), \mathbf{r_t} \Rightarrow \Gamma_A\{\mathbf{r_t}{:}|\tau_x|\}\{\mathcal{A}(x){:}|\tau_x|\}$

By lemma 34:

$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathbf{r_t}{:}|\tau_x|\}\{\mathcal{A}(x){:}|\tau_x|\} \vdash \mathbf{junk}\ \ \mathbf{r_t} \Rightarrow \Gamma_A\{\mathcal{A}(x){:}|\tau_x|\}$

By the good allocator assumption:

$\Gamma_A\{\mathcal{A}(x){:}|\tau_x|\} = |\Gamma, x\tau_x|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$

By inversion:

$\Psi; \Delta; \Gamma, x{:}\tau_x; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \rightsquigarrow I; F$
$\Psi; \Delta; \Gamma, x{:}\tau_x \vdash e : \tau\ \mathbf{exp}$

So by induction:
$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A\{\mathcal{A}(x){:}|\tau_x|\} \vdash I$ **ok**

17. Suppose $\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_C e : \tau \leadsto (S{:}I); F$ via the spill rule.

    By inversion:
    $\Psi; \Delta; \Gamma; \mathcal{A}', \sigma_1, \sigma_2 \vdash_C e : \tau \leadsto I; F$
    where $\mathcal{A}$' is a good allocator for e
    $\Psi; \Delta; \Gamma \vdash e : \tau$ **exp**

    By induction:
    $|\Psi| \vdash F : \Psi_F$
    $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_B \vdash I$ **ok**where $\Gamma_B = |\Gamma|^{\sigma_1 \circ \sigma_2}_{\mathcal{A}'}$

    By inversion:
    $|\Psi|; \Delta^{\rho_1,\rho_1}; |\Gamma|^{\sigma_1 \circ \sigma_2}_{\mathcal{A}} \vdash S \Rightarrow |\Gamma|^{\sigma_1 \circ \sigma_2}_{\mathcal{A}'}$

    So by lemma 29 (partial instruction sequence completion):
    $|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash S{:}I$ **ok**

■

### 8.3.6 Heap values, heaps, and programs

**Heap values**

$$\frac{\begin{array}{c}\Psi; \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k; x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{ret}} e : \tau \leadsto I; F \\ \text{Where } \mathcal{A} \text{ is a good allocator for e} \\ \text{and } \sigma_1 = (\mathbf{cont}(\overline{m + 2 * n})(\rho_1)(\rho_2)(|\tau|)) \triangleright_{32} \sigma_{32} \circ \sigma_{64} \circ \rho_1 \\ \text{and } \sigma_{32} = |\tau_1| \triangleright_{32} \cdots \triangleright_{32} |\tau_m| \triangleright_{32} \epsilon \\ \text{and } \sigma_{64} = |\phi_1| \triangleright_{64} \cdots \triangleright_{64} |\phi_n| \triangleright_{64} \epsilon \\ \text{and } \sigma_2 = \rho_2 \\ \text{and } l = \text{frmsz}(\mathcal{A}) \\ \text{and } \mathit{hval} = \texttt{code}[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k, \rho_1{:}ST, \rho_2{:}ST]|\texttt{Code}(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n)(\tau)| \\ \begin{array}{l} \texttt{salloc } l; \\ \mathbf{srmov}\ \mathcal{A}(x_1), \mathbf{sp}(l + 0); \\ \vdots \\ \mathbf{srmov}\ \mathcal{A}(x_m), \mathbf{sp}(l + m - 1); \\ \mathbf{srfmov}\ \mathcal{A}(z_1), \mathbf{sp}(l + m + 2 * 0); \\ \vdots \\ \mathbf{srfmov}\ \mathcal{A}(z_n), \mathbf{sp}(l + m + 2 * (n - 1)); \\ I \end{array}\end{array}}{\begin{array}{c}\Psi \vdash_h \texttt{code}_\tau[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k](x_1{:}\tau_1, \ldots, x_m{:}\tau_m)(z_1{:}\phi_1, \ldots, z_n{:}\phi_n).e : \\ \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k]\,\texttt{Code}(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n)(\tau) \leadsto \mathit{hval}; F\end{array}}$$

Since the only heap values in the **LIL** are code blocks, the heap value translation has only one inference rule, for translating code functions. Code functions are translated by translating their bodies with a good allocator, and with appropriate stack segment parameters. Recall that the stack segment parameters describe the stack below the current frame (which is managed by the allocator). Consequently, the stack segments in the code function translation reflect the calling convention on entry to the function: the top stack segment consists of an appropriate return continuation and the 32 and 64 bit arguments pushed onto the first stack segment type variable ($\rho_1$). The second stack segment, describing the section below the current handler, is simply the second stack type variable ($\rho_2$). Notice that the parameter of occurrence on the expression translation derivation indicates that the expression occurs in a return context and hence should return its value by popping the frame and calling the return continuation from the stack.

The result of translating the body is an instruction sequence $I$ and a heap fragment $F$ containing additional code blocks to be allocated at the top level. In order to complete the translation, it is necessary to wrap the instruction sequence $I$ with additional code to allocate the frame and to initialize the locations assigned to the argument variables from the locations assigned to the in-arguments by the calling convention. In principle the allocator may sometimes render this initialization code unnecessary by using variables directly from their in-argument slots, but for simplicity I do not attempt to perform this optimization here. The wrapped code sequence is then abstracted with respect to the free type variables (including the stack segment parameters) to turn it into a closed heap value.

**Theorem 23 (Soundness of the heap value translation)**
*If* $\quad \Psi \vdash \mathit{hval} : \tau\ \mathbf{hval}$
*and* $\quad \Psi \vdash_h \mathit{hval} : \tau \rightsquigarrow \mathit{hval}{:}F$
*then*
$\quad |\Psi| \vdash F : \Psi_F$
*and* $\quad |\Psi| \vdash \mathit{hval} : |\tau|\ \mathbf{hval}$

**Proof:** By construction.

By inversion:
$\mathcal{A}$ is a good allocator for e
$\Psi; \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k; x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n \vdash e : \tau\ \mathbf{exp}$
$\Psi; \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k; x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\mathtt{ret}} e : \tau \rightsquigarrow I; F$

And by theorem 15 and construction:
$\Delta^{\rho_1,\rho_2} \vdash \sigma_1 : ST$
$\Delta^{\rho_1,\rho_2} \vdash \sigma_2 : ST$

So by theorem 22:
$|\Psi| \vdash F : \Psi_F$
$|\Psi|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_A \vdash I\ \mathbf{ok}$
where $\Gamma_A = |x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$

It remains to be shown that:
$|\Psi| \vdash \mathit{hval} : |\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k]\,\mathtt{Code}(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n)(\tau)|\ \mathbf{hval}$

By lemma 29, it suffices to show that:

$$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_0 \vdash \texttt{salloc}\ l; \qquad \Rightarrow \Gamma_A$$
$$\begin{array}{l}
\textbf{srmov}\ \ \mathcal{A}(x_1), \textbf{sp}(l+0);\\
\vdots\\
\textbf{srmov}\ \ \mathcal{A}(x_m), \textbf{sp}(l+m-1);\\
\textbf{srfmov}\ \ \mathcal{A}(z_1), \textbf{sp}(l+m+2*0);\\
\vdots\\
\textbf{srfmov}\ \ \mathcal{A}(z_n), \textbf{sp}(l+m+2*(n-1));
\end{array}$$

where $\Gamma_0 = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2), \mathbf{r_t}{:}ns_{32}, \textbf{sp}{:}\sigma_1\}$
and $\Gamma_A = |x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2)\}$

By the salloc rule, it suffices to show that :

$$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_1 \vdash \textbf{srmov}\ \ \mathcal{A}(x_1), \textbf{sp}(l+0); \qquad \Rightarrow \Gamma_A$$
$$\begin{array}{l}
\vdots\\
\textbf{srmov}\ \ \mathcal{A}(x_m), \textbf{sp}(l+m-1);\\
\textbf{srfmov}\ \ \mathcal{A}(z_1), \textbf{sp}(l+m+2*0);\\
\vdots\\
\textbf{srfmov}\ \ \mathcal{A}(z_n), \textbf{sp}(l+m+2*(n-1));
\end{array}$$

where $\Gamma_1 = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2), \mathbf{r_t}{:}ns_{32}, \textbf{sp}{:}{ns_{32}}^{l} \triangleright_{32} \sigma_1\}$

Which follows by induction on $n+m$

- Suppose $n+m=0$

  Then by assumption:
  $\Gamma_A = |\bullet|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2)\}$

  And by the good allocator assumption:
  $|\bullet|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2)\}$
  $= \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2), \mathbf{r_t}{:}ns_{32}, \textbf{sp}{:}{ns_{32}}^{l} \triangleright_{32} \sigma_1\}$

- Suppose $m>0$. Then

  By induction:

  $$|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma_1 \vdash \textbf{srmov}\ \ \mathcal{A}(x_1), \textbf{sp}(l+0); \qquad \Rightarrow \Gamma'_A$$
  $$\begin{array}{l}
  \vdots\\
  \textbf{srmov}\ \ \mathcal{A}(x_m), \textbf{sp}(l+m-1);\\
  \textbf{srfmov}\ \ \mathcal{A}(z_1), \textbf{sp}(l+m+2*0);\\
  \vdots\\
  \textbf{srfmov}\ \ \mathcal{A}(z_{n-1}), \textbf{sp}(l+m+2*(n-2));
  \end{array}$$

  where $\Gamma'_A = |x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_{n-1}{:}\phi_{n-1}|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\textbf{Exnptr}(\sigma_2)\}$

  So by the definition of partial instruction sequences, it suffices to show that:
  $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma'_A \vdash \textbf{srfmov}\ \ \mathcal{A}(z_n), \textbf{sp}(l+m+2*(n-1)) \Rightarrow \Gamma_A$

  But note that by lemma 37:
  $|\Psi|;\Delta^{\rho_1,\rho_2};\Gamma'_A \vdash \textbf{srfmov}\ \ \mathcal{A}(z_n), \textbf{sp}(l+m+2*(n-1)) \Rightarrow \Gamma'_A\{\mathcal{A}(z_n){:}|\phi_n|$

And by the good allocator assumption:

$$\begin{aligned}\Gamma_A &= |x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_{n-1}{:}\phi_{n-1}, z_n{:}\phi_n|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\} \\ &= |x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_{n-1}{:}\phi_{n-1}|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}\{\mathcal{A}(z_n){:}|\phi_n|\} \\ &= \Gamma'_A\{\mathcal{A}(z_n){:}|\phi_n|\}\end{aligned}$$

- Suppose $m = 0$ and $n > 0$. Then

  By induction:

$$\begin{aligned}|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma_1 \vdash\ & \mathbf{srmov}\ \ \mathcal{A}(x_1), \mathbf{sp}(l+0); \qquad\qquad \Rightarrow \Gamma'_A \\ & \vdots \\ & \mathbf{srmov}\ \ \mathcal{A}(x_{m-1}), \mathbf{sp}(l+m-2);\end{aligned}$$

  where $\Gamma'_A = |x_1{:}\tau_1, \ldots, x_{m-1}{:}\tau_{m-1}, x_m{:}\tau_m|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}$

  So by the definition of partial instruction sequences, it suffices to show that:

$$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma'_A \vdash \mathbf{srmov}\ \ \mathcal{A}(x_m), \mathbf{sp}(l+m) \Rightarrow \Gamma_A$$

  But note that by lemma 36:

$$|\Psi|; \Delta^{\rho_1,\rho_2}; \Gamma'_A \vdash \mathbf{srmov}\ \ \mathcal{A}(x_m), \mathbf{sp}(l+m) \Rightarrow \Gamma'_A\{\mathcal{A}(x_m){:}|\tau_m|$$

  And by the good allocator assumption:

$$\begin{aligned}\Gamma_A &= |x_1{:}\tau_1, \ldots, x_{m-1}{:}\tau_{m-1}, x_m{:}\tau_m|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\} \\ &= |x_1{:}\tau_1, \ldots, x_{m-1}{:}\tau_{m-1}|_{\mathcal{A}}^{\sigma_1 \circ \sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2)\}\{\mathcal{A}(x_m){:}|\tau_m|\} \\ &= \Gamma'_A\{\mathcal{A}(x_m){:}|\tau_m|\}\end{aligned}$$

■

**Heaps**

$$\frac{}{\Psi \vdash \epsilon \leadsto \epsilon} \qquad \frac{\Psi[\ell{:}\tau] \vdash_h hval : \tau \leadsto hval'; F \quad \Psi[\ell{:}\tau] \vdash d \leadsto F'}{\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval \leadsto \ell{:}|\tau|.hval', F; F'}$$

The translation of heaps simply translates the individual heap bindings to produce a heap value and a new heap fragment. The heap value can then be bound to the assigned label, and incorporated with the new heap fragment into the rest of the re-written heap.

The heap soundness theorem is stated in two parts. Since the translation returns a heap fragment for each heap value, I first show that the result of the translation is well-formed as a heap fragment. The well-formedness of the translation of a closed well-formed heap then follows almost immediately.

**Theorem 24 (Soundness of the heap translation)**
*If* $\Psi \vdash d\ \mathbf{ok}$
*and* $\Psi \vdash d \leadsto F$
*then*
$|\Psi| \vdash F\ \mathbf{ok}\Psi_F$

**Proof:** (By induction on heaps)

- Suppose $d = \epsilon$.

By definition:
$\Psi \vdash \epsilon \rightsquigarrow \epsilon$

And by the heap fragment typing rule:
$|\Psi| \vdash \epsilon : \epsilon$

- Suppose $d = d', \ell{:}\tau \mapsto hval$

  By inversion:
  $\Psi[\ell{:}\tau] \vdash d' \ \mathbf{ok}$
  $\Psi[\ell{:}\tau] \vdash hval : \tau \ \mathbf{hval}$
  $\Psi[\ell{:}\tau] \vdash d \rightsquigarrow F'$
  $\Psi[\ell{:}\tau] \vdash_h hval : \tau \rightsquigarrow hval'; F$

  By induction:
  $|\Psi[\ell{:}\tau]| \vdash F' \ \mathbf{ok}\Psi'_F$

  By theorem 23:
  $|\Psi[\ell{:}\tau]| \vdash hval' : |\tau|$
  $|\Psi[\ell{:}\tau]| \vdash F \ \mathbf{ok}\Psi_F$

  By lemma 30:
  $|\Psi[\ell{:}\tau]| \vdash F; F' \ \mathbf{ok}\Psi_F, \Psi'_F$

  By the heap fragment formation rule:
  $|\Psi[\ell{:}\tau]| \vdash \ell{:}|\tau|.hval', F; F' \ \mathbf{ok}\Psi_F, \Psi'_F$

■

**Programs**

$$\frac{\begin{array}{c}\Psi(d) \vdash d \rightsquigarrow H' \quad \Psi(d); \bullet; \bullet; \mathcal{A}, \epsilon, \epsilon \vdash_{\texttt{halt}} e : \tau \rightsquigarrow I'; F \\ \text{Where } \mathcal{A} \text{ is a good allocator for e} \\ \text{and } H = F; H', \\ \qquad \ell_{\mathbf{ihandle}}{:}\ \mathbf{Exnhandler}(\epsilon) \\ \qquad\quad \texttt{mov}\ \mathbf{r_t}, \mathbf{r}_1; \\ \qquad\quad \texttt{halt}_{\texttt{Dyn}} \\ \text{and } R = \{\mathbf{r}_1 \mapsto ns_{32}, \mathbf{r}_2 \mapsto ns, \mathbf{r}_e \mapsto ns_{32}, \mathbf{r}_t \mapsto ns, \mathbf{f}_1 \mapsto ns_{64}, \mathbf{f}_2 \mapsto ns_{64}, \mathbf{f_t} \mapsto ns_{64}, \mathbf{sp} \mapsto \epsilon\} \\ \text{and } I = \texttt{malloc}\ \mathbf{r}_e[\mathbf{Exnhndler}(\epsilon), \epsilon]\langle \ell_{\mathbf{ihandle}}, \mathbf{sp}\rangle; I'\end{array}}{\bullet \vdash \texttt{letrec}\, d \,\texttt{in}\, e : \tau \rightsquigarrow H, R, I}$$

The translation of a **LIL** program is obtained by translating heap to obtain a **TILTAL** heap ($H'$), and translating the body of the program to obtain an instruction sequence ($I'$) and some additional heap blocks to be incorporated into the heap $F$). Note that the body of the program is translated with a halt occurrence parameter, indicating that the instruction sequence should be terminated with a `halt` instruction. In order to satisfy the calling conventions, it is necessary to add a final exception handler block to the heap ($\ell_{\mathbf{ihandle}}$) and allocate an appropriate exception frame containing it in $\mathbf{r}_e$. The initial register file contains nonsense in all of the general purpose registers, and an empty stack in the stack register.

**Theorem 25 (Soundness of the program translation)**
*If* $\quad \bullet \vdash \texttt{letrec}\, d\, \texttt{in}\, e : \tau$
*and* $\quad \bullet \vdash \texttt{letrec}\, d\, \texttt{in}\, e : \tau \rightsquigarrow (H, R, I)$
*then*

$$\bullet \vdash (H, R, I)\ \mathbf{ok}$$

**Proof:**

By inversion:
$\Psi(d) \vdash d\ \mathbf{ok}$
$\Psi(d) \vdash d \rightsquigarrow H'$

So by theorem 24:
$|\Psi(d)| \vdash H'\ \mathbf{ok}\Psi'_H$

By inversion:
$\Psi(d); \bullet; \bullet; \mathcal{A}, \epsilon, \epsilon \vdash_{\texttt{halt}} e : \tau \rightsquigarrow I'; F$
$\Psi(d); \bullet; \bullet \vdash e : \tau\ \mathbf{exp}$

And by the empty stack rule:
$\Delta^{\rho_1,\rho_2} \vdash \epsilon : ST$

So by theorem 22:
$|\Psi(d)| \vdash F : \Psi_F$
$|\Psi(d)|, \Psi_F; \Delta^{\rho_1,\rho_2}; \Gamma_0\{\mathbf{r}_e : \mathbf{Exnptr}(\epsilon)\} \vdash I'\ \mathbf{ok}$
where $\Gamma_0 = |\bullet|^{\epsilon}_{\mathcal{A}}$

By construction:
$$\ell_{\mathbf{ihandle}} : \mathbf{Exnhandler}(\epsilon) \vdash \ell_{\mathbf{ihandle}} : \mathbf{Exnhandler}(\epsilon)\ \mathbf{ok} \quad \begin{array}{l} \texttt{mov}\ \mathbf{r_t}, \mathbf{r}_1; \\ \texttt{halt}_{\texttt{Dyn}} \end{array}$$

And hence:
$$\ell_{\mathbf{ihandle}} : \mathbf{Exnhandler}(\epsilon), \Psi'_H, \Psi_F \vdash \begin{array}{l} F; H', \\ \ell_{\mathbf{ihandle}} : \mathbf{Exnhandler}(\epsilon) \\ \quad \texttt{mov}\ \mathbf{r_t}, \mathbf{r}_1; \\ \quad \texttt{halt}_{\texttt{Dyn}} \end{array} \mathbf{ok}$$

By the good allocator assumption:
$|\bullet|^{\epsilon}_{\mathcal{A}}\{\mathbf{r}_e : \mathbf{Exnptr}(\epsilon)\}$
$\quad = \{\mathbf{r}_1 : ns_{32}, \mathbf{r}_2 : ns, \mathbf{r}_e : \mathbf{Exnptr}(\epsilon), \mathbf{r}_t : ns, \mathbf{f}_1 : ns_{64}, \mathbf{f}_2 : ns_{64}, \mathbf{f_t} : ns_{64}, \mathbf{sp} : \epsilon\}$

By the malloc rule:
$\ell_{\mathbf{ihandle}} : \mathbf{Exnhndler}; \Delta; \Gamma_0 \vdash \texttt{malloc}\ \mathbf{r}_e[\mathbf{Exnhndler}(\epsilon), \epsilon]\langle \ell_{\mathbf{ihandle}}, \mathbf{sp} \rangle \Rightarrow \Gamma_0\{\mathbf{r}_e : \mathbf{Exnptr}(\epsilon)\}$

So by the instruction rule:
$|\Psi(d)|, \Psi(F), \ell_{\mathbf{ihandle}} : \mathbf{Exnhndler}; \Delta; \Gamma_0 \vdash I\ \mathbf{ok}$

So by the program rule :
$\vdash (H, R, I)\ \mathbf{ok}$

$\blacksquare$

## 8.4 The complete translation rules

**Small values** $\boxed{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\mathbf{32}} sv : \tau \leadsto sv'}$

$$\frac{}{\Psi; \Delta; \Gamma[x{:}\tau]; \mathcal{A}[x \to \mathbf{r}], \sigma_1, \sigma_2 \vdash_{32} x : \tau \leadsto \mathbf{r}}$$

$$\frac{}{\Psi; \Delta; \Gamma[x{:}\tau]; \mathcal{A}[x \to \mathbf{sp}(i)], \sigma_2, \sigma_2 \vdash_{32} x : \tau \leadsto \mathbf{sp}(i)}$$

$$\frac{}{\Psi[\ell{:}\tau]; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \ell : \tau \leadsto \ell}$$

$$\frac{}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} i : \texttt{Int} \leadsto i}$$

$$\frac{\Delta \vdash c \equiv \bigvee[\ldots, c_i, \ldots] : \mathrm{T}_{32} \qquad \Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : c_i \leadsto sv'}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \texttt{inj\_union}_c\, sv : c \leadsto \texttt{inj\_union}_{|c|}\, sv'}$$

$$\frac{\Delta \vdash \tau \equiv \texttt{Rec}[\kappa](c)(c_p) : \mathrm{T}_{32} \qquad \Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : c(\texttt{Rec}[\kappa]c)c_p \leadsto sv'}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \texttt{roll}_\tau\, sv : \tau \leadsto \texttt{roll}_{|\tau|}\, sv'}$$

$$\frac{\Delta \vdash \tau \equiv \texttt{Rec}[\kappa](c)(c_p) : \mathrm{T}_{32} \qquad \Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \tau \leadsto sv'}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \texttt{unroll}_\tau\, sv : c(\texttt{Rec}[\kappa]c)c_p \leadsto \texttt{unroll}_{|\tau|}\, sv'}$$

$$\frac{\Delta \vdash \tau \equiv \exists[\kappa](c') : \mathrm{T}_{32} \qquad \Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : c'c \leadsto sv'}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} \texttt{pack}\, sv\, \texttt{as}\, \tau\, \texttt{hiding}\, c : \tau \leadsto (\texttt{pack}[|\tau|]|c|)sv'}$$

$$\frac{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv : \forall[\kappa](c') \leadsto sv'}{\Psi; \Delta; \Gamma; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{32} sv[c] : c'c \leadsto sv'[|c|]}$$

**Float values** $\boxed{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{\mathbf{64}} fv:\phi\leadsto fv'}$

$$\frac{}{\Psi;\Delta;\Gamma[x_f{:}\phi];\mathcal{A}[x_f\rightarrow\mathbf{f}],\sigma_1,\sigma_2\vdash_{64} x_f:\phi\leadsto\mathbf{f}}$$

$$\frac{}{\Psi;\Delta;\Gamma[x_f{:}\phi];\mathcal{A}[x_f\rightarrow\mathbf{sp}(i)],\sigma_2,\sigma_2\vdash_{32} x_f:\phi\leadsto\mathbf{sp}(i)}$$

$$\frac{}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_2,\sigma_2\vdash_{32}\mathfrak{r}:\texttt{Float}\leadsto\mathfrak{r}}$$

**64 bit Instructions** $\boxed{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash \mathit{dest_{32}}\leftarrow \mathit{fopr}:\phi\leadsto S}$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{f_t}\leftarrow \mathit{fopr}:\phi\leadsto S}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{sp}(i)\leftarrow \mathit{fopr}:\phi\leadsto S;\\ \qquad\qquad\qquad\qquad\qquad\qquad\texttt{fswrite}\ \mathbf{sp}(i),\mathbf{f_t};\\ \qquad\qquad\qquad\qquad\qquad\qquad\mathbf{fjunk}\ \mathbf{f_t}\end{array}}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} fv:\phi\leadsto fv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{f}\leftarrow fv:\phi\leadsto\texttt{fmov}\,\mathbf{f_t},fv'}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_1:\texttt{Array}_{64}(\phi)\leadsto sv_1'\quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_2:\texttt{Int}\leadsto sv_2'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{f}\leftarrow\texttt{sub}_\phi(sv_1,sv_2):\phi\leadsto\texttt{sub}_{|\phi|}\,\mathbf{f},sv_1',sv_2'}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv:\texttt{Boxed}(\phi)\leadsto sv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{f}\leftarrow\texttt{unbox}\ sv:\phi\leadsto\texttt{floadr}\ \mathbf{f},sv'}$$

**Instructions** $\boxed{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash \mathit{dest_{32}}\leftarrow \mathit{opr}:\tau\leadsto S}$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{r_t}\leftarrow \mathit{opr}:\tau\leadsto S}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{sp}(i)\leftarrow \mathit{opr}:\tau\leadsto S;\\ \qquad\qquad\qquad\qquad\qquad\qquad\texttt{swrite}\ \mathbf{sp}(i),\mathbf{r_t};\\ \qquad\qquad\qquad\qquad\qquad\qquad\mathbf{junk}\ \mathbf{r_t}\end{array}}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv:\tau\leadsto sv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash\mathbf{r}\leftarrow sv:\tau\leadsto\texttt{mov}\,\mathbf{r},sv'}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \tau_1 \times \ldots \tau_n \rightsquigarrow sv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \texttt{select}^i\, sv : \tau_i \rightsquigarrow \begin{array}{l}\texttt{mov}\,\mathbf{r}, sv' \\ \texttt{loadr}\,\mathbf{r}, \mathbf{r}(i)\end{array}}$$

$$\frac{}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \texttt{dyntag}_c : \texttt{Dyntag}(c) \rightsquigarrow \texttt{dyntag}_{|c|}\ \mathbf{r}}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{64} fv : \texttt{Float} \rightsquigarrow fv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \texttt{box}\, fv : \texttt{Boxed}(\phi) \rightsquigarrow \texttt{malloc}_{|\phi|}\ \mathbf{r}, fv'}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv_1 : \tau_i \rightsquigarrow sv_i'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \langle sv_1,\ldots,sv_n\rangle : \tau_1 \times \ldots \times \tau_n \rightsquigarrow \texttt{malloc}\ \mathbf{r}, [|\tau_1|,\ldots,|\tau_n|]\langle sv_1',\ldots,sv_n'\rangle}$$

$$\frac{\begin{array}{c}|\Gamma|^{\epsilon}_{\mathcal{A}}(\mathbf{sp}) = \{\mathbf{r}_1{:}ns_{32}, \mathbf{r}_2{:}ns_{32}, \mathbf{r}_e{:}ns_{32}, \mathbf{r}_t{:}ns_{32}, \mathbf{f}_1{:}ns_{64}, \mathbf{f}_2{:}ns_{64}, \mathbf{sp} : \sigma_f\} \\ \Delta^{\rho_1,\rho_2} \vdash \sigma_f \equiv \underbrace{ns_{32} \triangleright_{32} \cdots \triangleright_{32} ns_{32}}_{m} \triangleright_{32} \underbrace{ns_{64} \triangleright_{64} \cdots ns_{64}}_{k} \triangleright_{64} \sigma' : ST \\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \texttt{Code}(\tau_0,\ldots,\tau_{m-1})(\phi_0,\ldots,\phi_{k-1}) \to \tau \rightsquigarrow sv' \\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv_i : \tau_i \rightsquigarrow sv_i' \quad i \in 0 \ldots m-1 \\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} fv_i : \phi_i \rightsquigarrow fv_i' \quad i \in 0 \ldots k-1\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \texttt{call}\ sv(sv_0,\ldots,sv_{m-1})(fv_0,\ldots,fv_{k-1}) : \tau \rightsquigarrow \\ \qquad \texttt{fswrite}\ \mathbf{sp}(m + 2*(k-1)), fv_{k-1}' \\ \qquad \vdots \\ \qquad \texttt{fswrite}\ \mathbf{sp}(m), fv_0' \\ \qquad \texttt{swrite}\ \mathbf{sp}(m-1), sv_{m-1}' \\ \qquad \vdots \\ \qquad \texttt{swrite}\ \mathbf{sp}(0), sv_0' \\ \qquad \textbf{junkregs} \\ \qquad \texttt{call}\, sv'[\sigma' \circ \sigma_1, \sigma_2] \\ \qquad \textbf{movj}\ \mathbf{r}, \mathbf{r_t}\end{array}}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv_1 : \texttt{Int} \rightsquigarrow sv_1' \quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv_2 : \tau \rightsquigarrow sv_2'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \texttt{array}_{\tau}(sv_1, sv_2) : \texttt{Array}_{32}(\tau) \rightsquigarrow \texttt{malloc}_{|\tau|}\ \mathbf{r}, sv_1', sv_2'}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \texttt{Int} \rightsquigarrow sv' \quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{64} fv : \phi \rightsquigarrow fv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathbf{r} \leftarrow \texttt{farray}(sv, fv) : \texttt{Farray}(\phi) \rightsquigarrow \texttt{fmalloc}_{|\phi|}\, \mathbf{r}, sv', fv'}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_1:\texttt{Array}_{32}(\tau)\rightsquigarrow sv_1' \quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_2:\texttt{Int}\rightsquigarrow sv_2'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash \mathbf{r}\leftarrow\texttt{sub}_{\tau}(sv_1,sv_2):\tau\rightsquigarrow \texttt{sub}_{|\tau|}\,\mathbf{r},sv_1',sv_2'}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_1:\texttt{Array}_{32}(\tau)\rightsquigarrow sv_1'\\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_2:\texttt{Int}\rightsquigarrow sv_2' \quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_3:\tau\rightsquigarrow sv_3'\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash \mathbf{r}\leftarrow\texttt{upd}_{\tau}(sv_1,sv_2,sv_3):\texttt{Unit}\rightsquigarrow \texttt{upd}\,sv_1',sv_2',sv_3'\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{malloc }\mathbf{r},[]\langle\rangle\end{array}}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_1:\texttt{Array}_{64}(\phi)\rightsquigarrow sv_1'\\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv_2:\texttt{Int}\rightsquigarrow sv_2' \quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{64} fv:\phi\rightsquigarrow fv'\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash \mathbf{r}\leftarrow\texttt{upd}_{\phi}(sv_1,sv_2,fv):\texttt{Unit}\rightsquigarrow \texttt{upd}\,sv_1',sv_2',fv'\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{malloc }\mathbf{r},[]\langle\rangle\end{array}}$$

**Expressions** $\boxed{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_C e:\tau\rightsquigarrow I;H}$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A}',\sigma_1,\sigma_2\vdash_C e:\tau\rightsquigarrow I;F\\ |\Psi|;\Delta^{\rho_1,\rho_1};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\vdash S\Rightarrow|\Gamma|_{\mathcal{A}'}^{\sigma_1\circ\sigma_2}\\ \text{where }\mathcal{A}\text{' is a good allocator for e}\end{array}}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_C e:\tau\rightsquigarrow(S{:}I);F}$$

$$\frac{\begin{array}{c}\Delta^{\rho_1,\rho_2}\vdash\sigma_1\equiv(\mathbf{cont}(\overline{m})(\sigma_1')(\sigma_2)(|\tau_r|))\triangleright_{32}\sigma_1':ST\\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv:\tau_r\rightsquigarrow sv' \quad \text{frmsz}(\mathcal{A})=n\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{\texttt{ret}} sv:\tau_r\rightsquigarrow \texttt{mov }\mathbf{r_t},sv';\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{sfree }n;\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{junkregs};\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{junkstack}\ 1\ldots m;\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{ret}\end{array}}$$

$$\frac{\begin{array}{c}|\Psi|;\Delta^{\rho_1,\rho_2};|\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\vdash sv_l:\Gamma\{\mathbf{r_t}{:}|\tau|\}\rightarrow 0\\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{32} sv:\tau\rightsquigarrow sv'\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2\vdash_{\texttt{jmp}\,sv_l} sv:\tau\rightsquigarrow \texttt{mov }\mathbf{r_t},sv'\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{jmp }sv_l\end{array}}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\epsilon,\epsilon\vdash_{32} sv:\tau_r\rightsquigarrow sv' \quad \text{frmsz}(\mathcal{A})=n}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\epsilon,\epsilon\vdash_{\texttt{halt}} sv:\tau_r\rightsquigarrow \texttt{mov }\mathbf{r_t},sv'\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{sfree }n\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{halt}_{\tau_r}\end{array}}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathcal{A}(x) \leftarrow \mathit{opr} : \tau_i \leadsto S \\ \Psi;\Delta;\Gamma,x{:}\tau_i;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e:\tau \leadsto I;F\end{array}}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}_\tau\, x = \mathit{opr}\,\mathtt{in}\, e:\tau \leadsto (S;I);F}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash \mathcal{A}(x_f) \leftarrow i : \phi_i \leadsto S \\ \Psi;\Delta;\Gamma,x_f{:}\phi_i;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e:\tau \leadsto I;F\end{array}}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}_\tau\, x_f = \mathit{fopr}\,\mathtt{in}\, e:\tau \leadsto (S;I);F}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \exists[\kappa][c] \leadsto sv' \\ \Delta,\alpha{:}\kappa;\Gamma,x{:}(c\alpha);\mathcal{A},\sigma_1,\sigma_2 \vdash_C e:\tau \leadsto I;F\end{array}}{\Psi;\Delta;\Gamma;\mathcal{A} \vdash_C \mathtt{let}[\alpha,x] = \mathtt{unpack}\,sv\,\mathtt{in}\,e:\tau \leadsto \begin{array}[t]{l}\mathtt{unpack}\ [\alpha,\mathbf{r_t}]sv'; \\ \mathbf{srmov}\ \ \mathcal{A}(x),\mathbf{r_t}; \\ \mathbf{junk}\ \ \mathbf{r_t}; \\ I; \\ F\end{array}}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \langle c_1,c_2\rangle : \kappa_1 \times \kappa_2 \\ \Psi;\Delta,;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e[c_1,c_2/\beta,\gamma] : \tau[c_1,c_2/\beta,\gamma] \leadsto I;F\end{array}}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\langle\beta,\gamma\rangle = c\,\mathtt{in}\,e:\tau \leadsto I;F}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \alpha : \kappa_1 \times \kappa_2 \\ \left.\begin{array}{l}\Psi;\Delta,\beta{:}\kappa_1,\gamma{:}\kappa_2; \\ \Gamma[\langle\beta,\gamma\rangle/\alpha]; \\ \mathcal{A},\sigma_1[\langle\beta,\gamma\rangle/\alpha],\sigma_2[\langle\beta,\gamma\rangle/\alpha]\end{array}\right\} \vdash_{C[\langle\beta,\gamma\rangle/\alpha]} e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha] \leadsto I;F\end{array}}{\Psi;\Delta,\alpha{:}\kappa_1\times\kappa_2;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\langle\beta,\gamma\rangle = c\,\mathtt{in}\,e:\tau \leadsto \begin{array}[t]{l}\mathtt{refine}\langle\beta,\gamma\rangle = |c|; \\ I; \\ F\end{array}}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \mathtt{fold}_{\mu j.\kappa}\, c' : \mu j.\kappa \\ \Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_1 \vdash_C e[c'/\beta] : \tau \leadsto I;F\end{array}}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\,\mathtt{fold}\,\beta = c\,\mathtt{in}\,e:\tau \leadsto I;F}$$

$$\frac{\begin{array}{c}\Delta,\alpha{:}\mu j.\kappa,\Delta' \vdash c \equiv \alpha : \mu j.\kappa \\ \left.\begin{array}{l}\Psi;\Delta,\beta{:}\kappa[\mu j.\kappa/j],\Delta'; \\ \Gamma[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha]; \\ \mathcal{A}, \\ \sigma_1[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha], \\ \sigma_1[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha]\end{array}\right\} \vdash_{C[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha]} e[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] : \tau[\mathtt{fold}_{\mu j.\kappa}\,\beta/\alpha] \leadsto I;F\end{array}}{\Psi;\Delta,\alpha{:}\mu j.\kappa,\Delta';\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\,\mathtt{fold}\,\beta = c\,\mathtt{in}\,e:\tau \leadsto \begin{array}[t]{l}\mathtt{refine}[\mathtt{fold}\,\beta]c; \\ I; \\ F\end{array}}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \mathtt{inj}_i^{+[\kappa_1,\ldots,\kappa_i,\ldots,\kappa_n]}\, c' : +[\kappa_1,\ldots,\kappa_i,\ldots,\kappa_n] \\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e[c'/\beta] : \tau \rightsquigarrow I; F\end{array}}{\Psi;\Delta;\Gamma \vdash_C \mathtt{let}_\tau\, \mathtt{inj}_i\, \beta = (c, sv)\, \mathtt{in}\, e : \tau \rightsquigarrow I; F}$$

$$\frac{\begin{array}{c}\Delta,\alpha{:}+[\kappa_1\ldots,\kappa_n],\Delta' \vdash c \equiv \alpha : +[\kappa_1\ldots\kappa_n] \quad \beta \notin \Delta,\Delta' \\ \left.\begin{array}{c}\Psi;\Delta,\beta{:}\kappa_i,\Delta'; \\ \Gamma[\mathtt{inj}_i\,\beta/\alpha]; \\ \mathcal{A},\sigma_1[\mathtt{inj}_i\,\beta/\alpha],\sigma_2[\mathtt{inj}_i\,\beta/\alpha]\end{array}\right\} \vdash_{C[\mathtt{inj}_i\,\beta/\alpha]} e[\mathtt{inj}_i\,\beta/\alpha] : \tau[\mathtt{inj}_i\,\beta/\alpha] \rightsquigarrow I; F \\ \Psi;\Delta,\beta{:}\kappa_j,\Delta';\Gamma[\mathtt{inj}_j\,\beta/\alpha];\mathcal{A},\sigma_1[\mathtt{inj}_j\,\beta/\alpha],\sigma_2[\mathtt{inj}_j\,\beta/\alpha] \vdash_{32} sv[\mathtt{inj}_j\,\beta/\alpha] : \mathtt{Void} \rightsquigarrow sv' \\ j \in 1\ldots i-1, i+1\ldots n\end{array}}{\Psi;\Delta,\alpha{:}+[\kappa_1,\ldots,\kappa_n],\Delta' \vdash_C \mathtt{let}_\tau\, \mathtt{inj}_i\, \beta = (c, sv)\, \mathtt{in}\, e : \tau \rightsquigarrow \begin{array}{l}\mathtt{refine}[\mathtt{inj}_i\,\beta]|c|, sv'; \\ I; \\ F\end{array}}$$

$$\frac{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \mathtt{Dyn} \rightsquigarrow sv'}{\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \mathtt{let}\, x = \mathtt{raise}_\tau\, sv\, \mathtt{in}\, e : \tau \rightsquigarrow \begin{array}{l}\mathtt{mov}\, \mathbf{r}_1, sv \\ \mathtt{loadr}\, \mathbf{r_t}, \mathbf{r}_e(1) \\ \mathtt{mov}\, \mathbf{sp}, \mathbf{r_t} \\ \mathtt{loadr}\, \mathbf{r_t}, \mathbf{r}_e(0) \\ \mathtt{jmp}\, \mathbf{r_t}\end{array}}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \texttt{Dyn} \leadsto sv' \quad \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv_1 : \texttt{Dyntag}(\tau_1) \leadsto sv' \\ \Psi;\Delta;\Gamma,x_1{:}\times[\texttt{Dyntag}(\tau_1),\tau_1];\mathcal{A},\sigma_1,\sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_1 : \tau_x \leadsto I_1;F_1 \\ \Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_2 : \tau_x \leadsto I_2;F_2 \\ \Psi;\Delta;\Gamma,x{:}\tau_x;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e : \tau \leadsto I;F \\ |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} = \Gamma_A\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \texttt{let}\,x = \texttt{dyncase}(sv)(sv_1 \Rightarrow x_1.e_1, \_ \Rightarrow e_2)\,\texttt{in}\,e : \tau \leadsto \\ \quad \texttt{mov}\,\mathbf{r_t}, sv' \\ \quad \texttt{brdyn}\ \mathbf{r_t}, sv_1', \ell_1[(\Pi_1\Delta),\sigma_1,\sigma_2] \\ \quad \textbf{junk}\ \mathbf{r_t} \\ \quad I_2 \\ \ell_1{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}\times[\texttt{Dyntag}(|\tau_1|),|\tau_1|]] \\ \quad \textbf{srmov}\,\mathcal{A}(x_1),\mathbf{r_t} \\ \quad \textbf{junk}\ \mathbf{r_t} \\ \quad I_1 \\ \ell_{\mathbf{end}}{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_x|] \\ \quad \textbf{srmov}\ \mathcal{A}(x),\mathbf{r_t} \\ \quad \textbf{junk}\ \mathbf{r_t} \\ \quad I; \\ F; \\ F_1; \\ F_2\end{array}}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\epsilon,\sigma_3 \vdash_{\texttt{jmp}\,\ell_{\mathbf{post}}[(\Pi_1\Delta),\epsilon,\sigma_h]} e_1 : \tau_x \rightsquigarrow I_1; F_1 \\ \Psi;\Delta;\Gamma,x_2{:}\,\texttt{Dyn};\mathcal{A},\sigma_1,\sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_f\circ\sigma_1,\sigma_2]} e_2 : \tau_x \rightsquigarrow I_2; F_2 \\ \Psi;\Delta;\Gamma,x{:}\tau_x;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e : \tau \rightsquigarrow I; F \\ \Gamma_A = \{\mathbf{r}_1{:}ns_{32},\mathbf{r}_2{:}ns_{32},\mathbf{r}_t{:}ns_{32},\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2),\mathbf{sp}{:}\sigma_f\circ\sigma_1\circ\sigma_2,\mathbf{f}_1{:}ns_{64},\mathbf{f}_2{:}ns_{64}\} \\ \text{where} \quad \Gamma_A = |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_2) \\ \text{and} \quad \Gamma_B = |\Gamma|_{\mathcal{A}}^{\sigma_3}\{\mathbf{r}_e{:}\,\mathbf{Exnptr}(\sigma_3) \\ \text{and} \quad \sigma_f = |\Gamma|_{\mathcal{A}}^{\epsilon}(\mathbf{sp}) \\ \text{and} \quad \sigma_3 = \mathbf{Exnptr}(\sigma_2) \triangleright_{32} \sigma_f\circ\sigma_1\circ\sigma_2 \\ \text{and} \quad \sigma_h = \sigma_f\circ\sigma_3\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \texttt{let}\,x = \texttt{handle}_{\tau_x}(e_1,x_2.e_2)\,\texttt{in}\,e : \tau \rightsquigarrow \\ \quad\quad \texttt{push}\ \mathbf{r}_e \\ \quad\quad \textbf{stackcopy}\ (\mathbf{Exnptr}(\sigma_2))\triangleright_{32}\sigma_f \\ \quad\quad \texttt{sfree}\ 1 \\ \quad\quad \texttt{malloc}\ \mathbf{r}_e[\mathbf{Exnhndler}(\sigma_h),\sigma_h]\langle\ell_{\mathbf{handle}}[\Pi_1\Delta,\rho_1,\rho_2],\mathbf{sp}\rangle \\ \quad\quad I_1 \\ \quad \ell_{\mathbf{handle}}{:}\forall[\Delta,\rho_1,\rho_2].\{\mathbf{r}_1{:}\,\texttt{Dyn},\mathbf{r}_2{:}ns_{32},\mathbf{r}_t{:}ns_{32},\mathbf{r}_e{:}ns_{32},\mathbf{sp}{:}\sigma_h,\mathbf{f}_1{:}ns_{64},\mathbf{f}_2{:}ns_{64}\} \\ \quad\quad \textbf{srmov}\ \mathcal{A}(x_2),\mathbf{r}_1 \\ \quad\quad \texttt{sfree}(\text{frmsz}(\mathcal{A})) \\ \quad\quad \texttt{pop}\ \mathbf{r}_e \\ \quad\quad \textbf{junk}\ \mathbf{r}_1 \\ \quad\quad I_2 \\ \quad \ell_{\mathbf{post}}{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_B[\mathbf{r_t}{:}|\tau_x|] \\ \quad\quad \texttt{sfree}(\text{frmsz}(\mathcal{A})) \\ \quad\quad \texttt{pop}\ \mathbf{r}_e \\ \quad\quad \texttt{jmp}\ \ell_{\mathbf{end}}[\Pi_1\Delta,\sigma_1,\sigma_2] \\ \quad \ell_{\mathbf{end}}{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_x|] \\ \quad\quad \textbf{srmov}\ \mathcal{A}(x),\mathbf{r_t} \\ \quad\quad \textbf{junk}\ \mathbf{r_t} \\ \quad\quad I; \\ \quad F; \\ \quad F_1; \\ \quad F_2\end{array}}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_{32} sv : \bigvee[\tau_1,\ldots,\tau_n] \rightsquigarrow sv' \\ \Delta \vdash \tau_i \equiv \texttt{Tag}(i) : \mathrm{T}_{32} \quad i \in 1\ldots(j-1) \qquad \Delta \vdash \tau_i \equiv \times[\texttt{Tag}(i),\tau_i'] : \mathrm{T}_{32} \quad i \in j\ldots n \\ \Psi;\Delta;\Gamma,x_i{:}\tau_i;\mathcal{A},\sigma_1,\sigma_2 \vdash_{\texttt{jmp}\,\ell_{\mathbf{end}}[(\Pi_1\Delta),\sigma_1,\sigma_2]} e_i : \tau \rightsquigarrow I_i;F_i \quad i \in 1\ldots n \\ \Psi;\Delta;\Gamma,x{:}\tau;\mathcal{A},\sigma_1,\sigma_2 \vdash_C e : \tau_e \rightsquigarrow I;F \\ |\Gamma|_{\mathcal{A}}^{\sigma_1\circ\sigma_2} = \Gamma_A\end{array}}{\begin{array}{l}\Psi;\Delta;\Gamma;\mathcal{A},\sigma_1,\sigma_2 \vdash_C \texttt{let}\,x = \texttt{case}_\tau(sv)(x_0.e_0,\ldots,x_n.e_n)\,\texttt{in}\,e : \tau_e \rightsquigarrow \\ \qquad \begin{array}{l} \quad \texttt{mov}\,\mathbf{r_t}, sv' \\ \quad \texttt{brtag}_0\,\mathbf{r_t}, \ell_0[(\Pi_1\Delta),\sigma_1,\sigma_2] \\ \quad \vdots \\ \quad \texttt{brtag}_{k-1}\,\mathbf{r_t}, \ell_{k-1}[(\Pi_1\Delta),\sigma_1,\sigma_2] \\ \quad \texttt{brtgd}_k\,\mathbf{r_t}, \ell_k[(\Pi_1\Delta),\sigma_1,\sigma_2] \\ \quad \vdots \\ \quad \texttt{brtag}_{n-1}\,\mathbf{r_t}, \ell_{n-1}[(\Pi_1\Delta),\sigma_1,\sigma_2] \\ \quad \texttt{mov}\,\mathbf{r_t}, \texttt{forgetunion}\,\mathbf{r_t} \\ \quad \texttt{jmp}\,\ell_n[(\Pi_1\Delta),\sigma_1,\sigma_2] \\ \ell_0{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_0|] \\ \quad \mathbf{srmov}\,\mathcal{A}(x_0),\mathbf{r_t} \\ \quad \mathbf{junk}\ \mathbf{r_t} \\ \quad I_0 \\ \ell_1{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_1|] \\ \vdots \\ \ell_n{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau_n|] \\ \quad \mathbf{srmov}\,\mathcal{A}(x_n),\mathbf{r_t} \\ \quad \mathbf{junk}\ \mathbf{r_t} \\ \quad I_n \\ \ell_{\mathbf{end}}{:}\forall[|\Delta|,\rho_1{:}ST,\rho_2{:}ST].\Gamma_A[\mathbf{r_t}{:}|\tau|] \\ \quad \mathbf{srmov}\ \mathcal{A}(x),\mathbf{r_t} \\ \quad \mathbf{junk}\ \mathbf{r_t} \\ \quad I; \\ F; \\ F_1; \\ \vdots \\ F_n \end{array}\end{array}}$$

**Heap values** $\boxed{\Psi \vdash_h hval : \tau \rightsquigarrow hval; H}$

$$\dfrac{\begin{array}{c}\Psi; \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k; x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n; \mathcal{A}, \sigma_1, \sigma_2 \vdash_{\texttt{ret}} e : \tau \rightsquigarrow I; F \\ \text{Where } \mathcal{A} \text{ is a good allocator for e} \\ \text{and } \sigma_1 = (\mathbf{cont}(\overline{m + 2 * n})(\rho_1)(\rho_2)(|\tau|)) \triangleright_{32} \sigma_{32} \circ \sigma_{64} \circ \rho_1 \\ \text{and } \sigma_{32} = |\tau_1| \triangleright_{32} \cdots \triangleright_{32} |\tau_m| \triangleright_{32} \epsilon \\ \text{and } \sigma_{64} = |\phi_1| \triangleright_{64} \cdots \triangleright_{64} |\phi_n| \triangleright_{64} \epsilon \\ \text{and } \sigma_2 = \rho_2 \\ \text{and } l = \text{frmsz}(\mathcal{A}) \\ \text{and } hval = \texttt{code}[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k, \rho_1{:}ST, \rho_2{:}ST] |\texttt{Code}(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n)(\tau)| \\ \begin{array}{l} \texttt{salloc } l; \\ \mathbf{srmov}\ \ \mathcal{A}(x_1), \mathbf{sp}(l + 0); \\ \vdots \\ \mathbf{srmov}\ \ \mathcal{A}(x_m), \mathbf{sp}(l + m - 1); \\ \mathbf{srfmov}\ \ \mathcal{A}(z_1), \mathbf{sp}(l + m + 2 * 0); \\ \vdots \\ \mathbf{srfmov}\ \ \mathcal{A}(z_n), \mathbf{sp}(l + m + 2 * (n - 1)); \\ I \end{array}\end{array}}{\begin{array}{c}\Psi \vdash_h \texttt{code}_\tau[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k](x_1{:}\tau_1, \ldots, x_m{:}\tau_m)(z_1{:}\phi_1, \ldots, z_n{:}\phi_n).e : \\ \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k]\, \texttt{Code}(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n)(\tau) \rightsquigarrow hval; F\end{array}}$$

**Heaps** $\boxed{\Psi \vdash d \rightsquigarrow H}$

$$\dfrac{}{\Psi \vdash \epsilon \rightsquigarrow \epsilon}$$

$$\dfrac{\Psi[\ell{:}\tau] \vdash_h hval : \tau \rightsquigarrow hval'; F \quad \Psi[\ell{:}\tau] \vdash d \rightsquigarrow F'}{\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval \rightsquigarrow \ell{:}|\tau|.hval', F; F'}$$

**Programs** $\boxed{\vdash p : \tau \rightsquigarrow P}$

$$\dfrac{\begin{array}{c}\Psi(d) \vdash d \rightsquigarrow H' \quad \Psi(d); \bullet; \bullet; \mathcal{A}, \epsilon, \epsilon \vdash_{\texttt{halt}} e : \tau \rightsquigarrow I'; F \\ \text{Where } \mathcal{A} \text{ is a good allocator for e} \\ \text{and } H = F; H', \\ \begin{array}{l} \ell_{\mathbf{ihandle}}{:}\, \mathbf{Exnhandler}(\epsilon) \\ \quad \texttt{mov } \mathbf{r_t}, \mathbf{r}_1; \\ \quad \texttt{halt}_{\texttt{Dyn}} \end{array} \\ \text{and } R = \{\mathbf{r}_1 \mapsto ns_{32}, \mathbf{r}_2 \mapsto ns, \mathbf{r}_e \mapsto ns_{32}, \mathbf{r}_t \mapsto ns, \mathbf{f}_1 \mapsto ns_{64}, \mathbf{f}_2 \mapsto ns_{64}, \mathbf{f_t} \mapsto ns_{64}, \mathbf{sp} \mapsto \epsilon\} \\ \text{and } I = \texttt{malloc } \mathbf{r}_e[\mathbf{Exnhndler}(\epsilon), \epsilon]\langle \ell_{\mathbf{ihandle}}, \mathbf{sp}\rangle; I'\end{array}}{\bullet \vdash \texttt{letrec}\, d\, \texttt{in}\, e : \tau \rightsquigarrow H, R, I}$$

# Chapter 9

# Implementation

In the previous chapters I developed a series of translations mapping programs in an idealized version of the original **TILT** internal language (the **MIL**) down to an idealized typed assembly language (**TILTAL**) and proved the soundness of these translations. In this chapter I describe a new backend that I have implemented in the **TILT** compiler implementing a concrete version of these translations.

The overall structure of the new backend can be seen in figure 9.1. The implementations of the translations generally follow quite closely on the formal description, and the **LIL** language as implemented is almost exactly the same as the formal version. The implementation differs most significantly from the formal presentation in that its final target is the **TALx86** language [MCG$^+$99, GM00]. Unlike the RISC-like **TILTAL** language presented in chapter 7, **TALx86** is specialized to the x86 architecture. Moreover, **TALx86** uses numerous additional technologies (such as alias types [WM01]) that I do not attempt to account for in my formal presentation. The **TALx86** language was chosen as the target because of its the numerous tools already existing for it (such as an assembler, typechecker, and linker) which I was able to use with minimal modification.

In this chapter, I describe the individual passes added to the compiler as part of my implementation work and discuss briefly the important differences between the implementation and the theoretical presentation. In order to demonstrate the practicality of my approach, I present some measurements to quantify the size of the typed binaries and their runtime performance as compared to other compilers.

## 9.1 Singleton Elimination

A key issue in typed compilation is controlling the size of the intermediate forms of programs. Type annotations on internal representation terms quickly come to dominate the overall representation size to the point that representing and traversing the type information becomes by far the dominant performance issue. However, there is in general a great deal of redundancy in this type information which can be exploited to eliminate or reduce this problem.

A number of mechanisms have been suggested [Sha97, PCHS00, GM00] to attempt to deal with this problem. In the **TILT** compiler, the **MIL** as implemented controls type sizes by using *singleton kinds* to provide a form of type definition [PCHS00, SH99] intrinsic to the language. Types are bound to variables using a derived `let` form so that the type can be referred to subsequently via its name. Compiler passes such as common sub-expression elimination find redundant uses of types

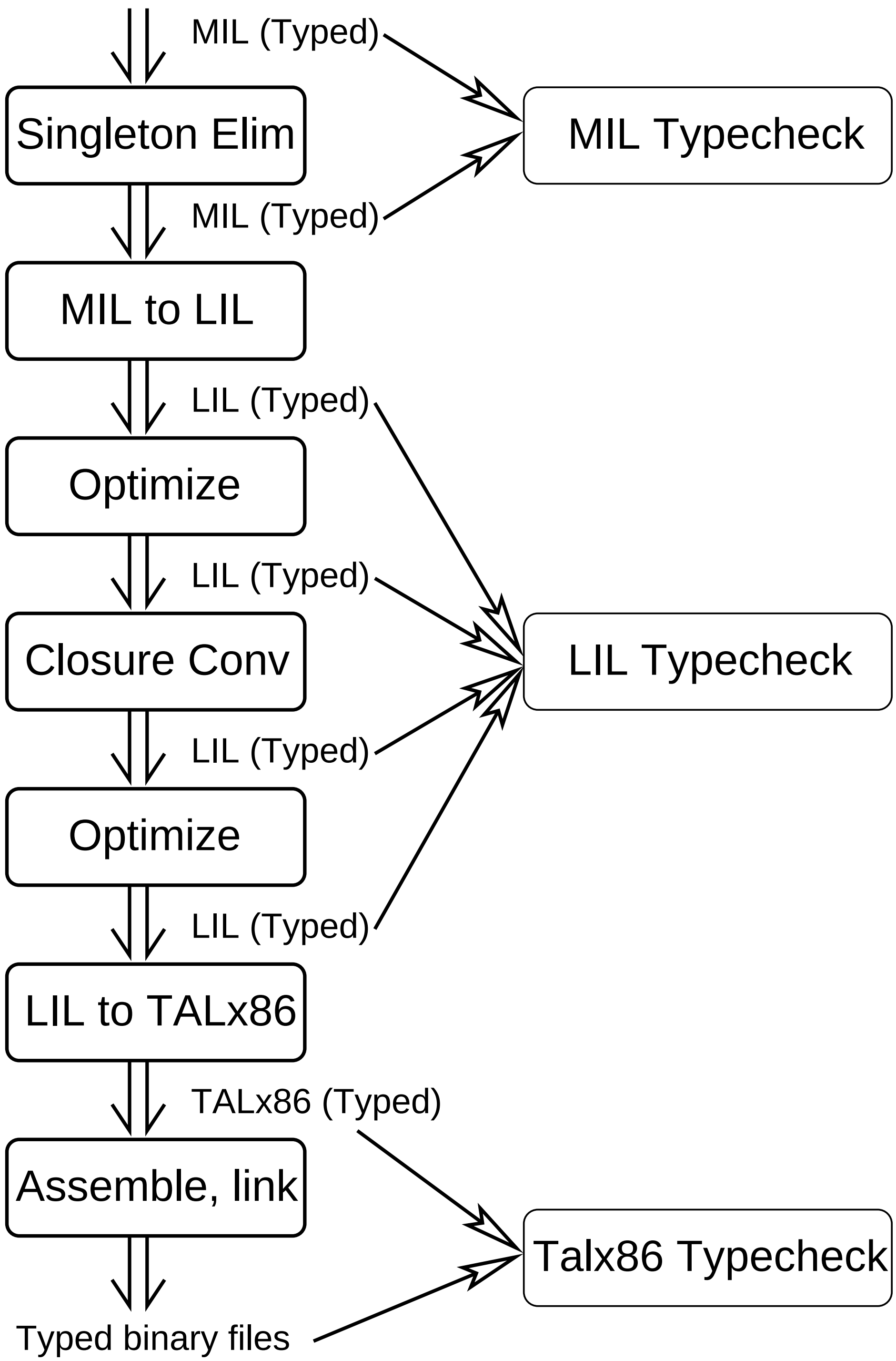



**Figure 9.1:** Structure of the certifying **TILT** backend

and reduce them to a single binding.

The type theory of singleton kinds has been greatly simplified by recent work. However, rather than attempt to reconcile this type theory with the LX type theory used for type-analysis, I chose for the purposes of the formal translation to use a singleton-free form of the **MIL**, and consequently to rely in the implementation of the new backend on other technologies to control the size of type information. Since the **TILT** compiler produces internal representations using singletons, the first additional pass to be added to the compiler was a singleton elimination pass.

Crary [Cra00] showed that the eliminability of singleton kinds follows from the completeness of the equivalence algorithm given by Stone and Harper [SH99] and gave an algorithm for computing the singleton free version of a term. The implementation of singleton elimination in the compiler follows Crary's algorithm almost exactly with one minor changes in order to avoid problems with type sizes, as explained below.

### 9.1.1 Preserving sharing of types

It would in theory have been possible to have implemented singleton elimination concurrently with the translation from the **MIL** to the **LIL** language. Since the **LIL** language has its own mechanisms for preserving sharing (discussed below in section 9.6), the result would have been a translation which neither traversed types repeatedly, nor generated internal forms with excess redundant type information. However, doing these two translations simultaneous seemed likely to be difficult and error-prone. Moreover, the singleton elimination phase introduces a large amount of additional constructor data that benefits from being exposed to the **MIL** optimization passes. Eliminating singletons concurrently with the translation to **LIL** would have prevented this.

For these reasons, singleton elimination was implemented in **TILT** in two phases. The early phase eliminates all singletons except uses which occur via the derived `let` form (an immediate application of a type lambda to a type in the underlying calculus). Since the derived elimination rule for the `let` construct is simple substitution, it is trivial to postpone the elimination of the derived `let` type construct until the translation to **LIL**, providing the **MIL** optimizer with an opportunity to improve the types emitted by the elimination phase[1]. This two phase approach allows singleton elimination to be done early in the compilation process while still preserving a compact representation using the `let` definitional mechanism until translation to **LIL**.

### 9.1.2 Optimizations for singleton elimination

#### Eta reduction

The singleton elimination pass introduces many new types that turn out to be eta-expansions of variables. It is straightforward to instrument the singleton elimination pass to catch many of these eta-expansions in order to perform the eta-reduction in place. However, even with this improvement on the original algorithm, singleton elimination still produces a fair number of eta-redices. Consequently, performing eta-reduction after singleton elimination is an important optimization to avoid redundant representation of types, as well as to reduce the size of the intermediate representation.

[1] An alternative way of viewing this implementation choice is as mapping into a calculus which can be given two different (but presumably equivalent) theories: one which provides a `let` form derived via singleton kinds, and one which provides a primitive `let` form with a substitution semantics. The equivalence of the two theories would have to be shown, however.

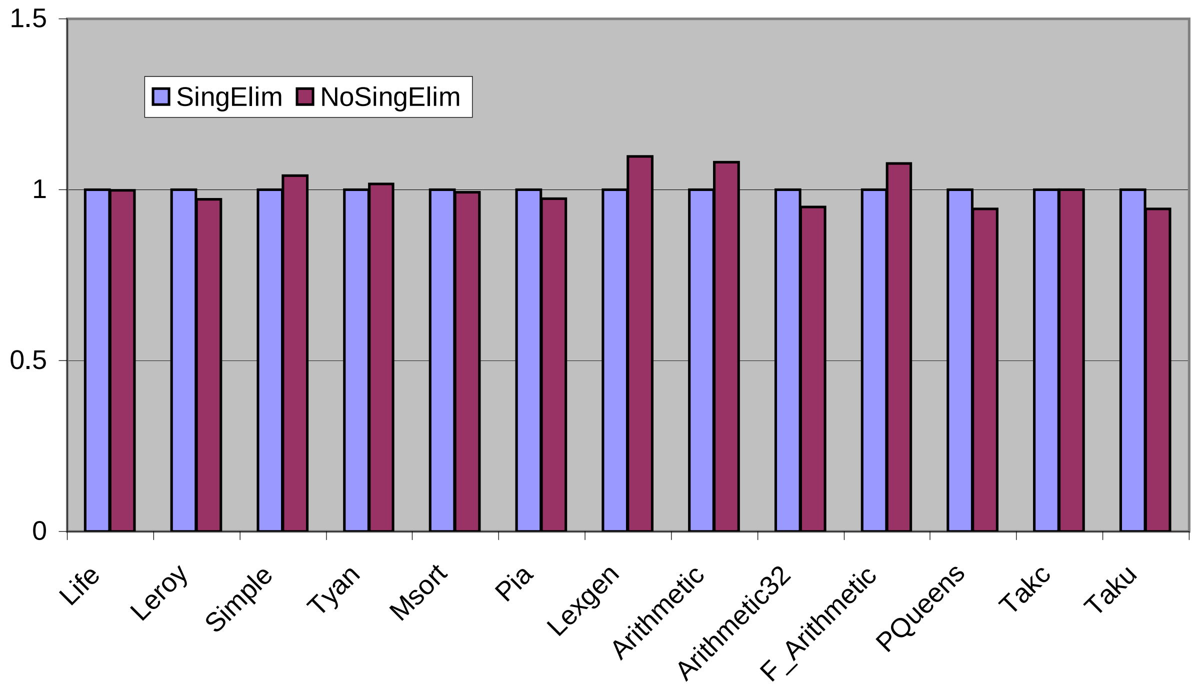



**Figure 9.2:** Benchmark timings with singleton elimination (normalized to timings without singleton elimination).

### Beta reduction

In addition to eta-redices, singleton elimination produces numerous beta-redices in the forms of projections from known records or applications of known functions. Eliminating the former is always a desired optimization for code in named-form since it can only reduce code size and eliminates an address calculation and a memory fetch. Eliminating the latter is a speculative optimization since the function may be called more than once. Consequently, even if it were practical to do these optimizations concurrently with singleton elimination, it is almost certainly preferable to leave this up to the general optimization and inlining code which must deal with these issues in any case.

### Common sub-expression elimination

Finally, singleton elimination may produce several copies of equivalent types. It is therefore valuable to do common sub-expression elimination (CSE) on the resulting code. It is particularly important after singleton elimination that CSE unify alpha-equivalent type functions, since it is not uncommon for many of the generated types to be simple alpha-variants.

## 9.1.3 Runtime behavior

Since types must be represented as data in the **TILT** compiler, it is important to consider the effect of singleton elimination on the runtime behavior of programs. While this information is not easily available for the certifying backend since it relies intrinsically on singleton elimination, code can be generated for the untyped Sparc backend using either normal or singleton-free internal representations. Once singletons are eliminated from the internal representations of the program, the remaining **MIL** optimization passes can be run as usual, since they introduce no uses of singleton kinds except in the form of the derived `let`. For the most part these passes behave identically when targeting both the untyped and the typed backends, up until the point where the translation to the next intermediate language takes place.

Figure 9.2 shows that across a wide range of benchmarks, the singleton-free code generally performs similarly to the non-singleton eliminated code. There was almost no difference in object code size between the two generated executables.

I speculate that the occasional improvement in the runtime behavior can be attributed to the singleton-elimination phase acting as a rudimentary form of cross-module inlining, by extracting all possible definitional information about abstract types from their kinds, and exposing that information to the optimizer. In principle, an optimization pass could be implemented to perform a similar function without actually doing singleton elimination.

## 9.2 MIL to LIL

The implemented translation of **MIL** code into **LIL** code follows the formal translation from chapter 5 almost exactly in structure. The only theoretically significant change is that in addition to the vararg and array optimizations presented in chapter 5, the implemented version performs a related representation optimization for sums. This optimization adds no significant complication to the translation, merely requiring that the encodings of types distinguish between those types which are inhabited by heap pointers and those which are inhabited by other values. This distinction is used to take advantage of the fact that pointers are never small integers. Sums with exactly one value carrying arm can be represented specially when the carried value is a pointer: there is no need to box and tag the pointer since it can always be distinguished from small integer tags.

The **MIL** intermediate representation which is passed to the translator is singleton-free, except with respect to let-bound definitions as discussed above. Some care is required in the implementation to avoid expanding out type definitions indiscriminately, and also to avoid unnecessary work. For example, consider a **MIL** term of the form `let a = c in e` The bound variable $a$ may potentially occur many times in the body of $e$. Moreover, it is not syntactically apparent whether $a$ is used as data (and hence requires static encoding and dynamic encoding) or as a classifier (and hence requires static encoding and interpretation), or as both, or neither. The definition of singleton elimination tells us that the correct translation of this term will be equivalent to the translation of $e[c/a]$ (since this is the intended meaning of the residual let binding). However, simply expanding out the definition via substitution and translating the resulting terms is unacceptably inefficient, both in terms of the size of the resulting intermediate representation and in terms of the cost of repeated translation of the type. The earlier compiler passes will have ensured that any such definition remaining is used at least once: often it will be the case that the binding is used many times (since similar bindings are unified via common-sub-expression elimination).

The implementation avoids the cost of re-traversing these bindings by maintaining a mapping from **MIL** type variables to memoized thunks that when forced, compute the static or dynamic encodings of variables as needed. In this way, no type binding is ever translated more than once as a type and once as a constructor. Moreover, the memoization ensures that all of the occurrences of the variable will be replaced by the same physically shared translation, preventing an explosion in the representation size.

## 9.3 Closure Conversion

Functions in the **LIL** intermediate forms produced by the translation from **MIL** code are still lexically nested and may refer to variables bound in enclosing scopes. In order to provide an efficient compiled implementation, a commonly used strategy is to replace uses of functions (evaluated via substitution) with uses of closed code and environments. Environments preserve the values of free variables, and are passed as additional arguments to code at runtime. Functions in the high level language become pairs of code pointers and environments in the underlying machine.

### 9.3.1 Closure conversion strategies

Numerous strategies have been proposed for handling closure conversion in a typed setting [CWM98, MWCG97, MMH96, MH98]. I briefly summarize the two main axes of variation from an implementation standpoint, and refer the reader to Morrisett and Harper [MH98] for a detailed analysis of the different approaches and their typing properties.

The first source of variation has to do with whether recursion is implemented via a primitive notion of recursive code, or via the introduction of recursive closures. In the recursive code approach, code functions are permitted to recursively refer to themselves and other code functions in the same nest within their own scope. In the recursive closure approach, all code is non-recursive. Closures however, may refer to back to themselves.

Many recursive closure implementations are based upon the idea of backpatching, in which the environment record is initialized to contain a pointer to the closure into which it will eventually be written. While simple and expedient to implement, this approach requires either a fairly powerful type system to type the resulting code, or else the use of ad-hoc "dummy closures" to stand in for the recursive reference in the initially allocated environment. In the interest of avoiding adding extra complexity to the problem, I chose to avoid this approach.

Another approach to implementing recursion is to parameterize code functions over their entire closure, instead of just the environment portion. Since the closure contains the code pointer to which it is being passed, this approach is sometimes referred to as the self-application approach.

Self-application is a fairly elegant approach to solving the problem of closure converting recursive code. However, it would require a fair bit of re-tooling in the existing TILT compiler in order to implement. In particular, the **MIL** type system does not support recursive types at higher kinds, which would seem to be necessary to efficiently support the self application semantics.

For the sake of simplicity in the implementation of closure conversion then, I chose to implement the relatively straightforward recursive code approach to closure conversion. Under this approach, mutually recursive functions nests become mutually recursive code nests, parameterized over a common environment. Functions become existentially packed tuples containing the code pointer and the environment of the function. At call sites, the code pointer and environment are projected out of the tuple, and the former is applied to the latter, along with the original arguments of the function.

### 9.3.2 Recursive code closure conversion

There are two well-known disadvantages to the recursive code approach to closure conversion.

The first is that since each function must be able to create the environment of each of its callees, the environment of each function in the nest must contain the transitive closure of all of the free variables of all of the functions in the nest (assuming that nests have been reduced to strongly connected components). In practice, this means that all of the functions in the nest share one environment, which is consequently larger than the individual environments might be. This is somewhat offset by the fact that only one environment per nest need be allocated.

The second, and potentially more serious drawback of the recursive code approach is that an escaping occurrence of a function within its own body requires the closure to be reconstructed on each invocation. For functions implementing loops, this can obviously be problematic. It is important to note however that it is *not* necessary to re-allocate the environment on each invocation: only the two element closure. Moreover, this particular idiom is relatively rare: most recursive functions do not escape within their body. Nonetheless, this remains a serious disadvantage of this approach.

The advantage of this approach is that it simple to implement, and moreover is relatively simple to implement efficiently. Most applications of local functions (including all recursive calls) can be implemented as calls directly to code, without reference to closures. Consequently, for most non-escaping functions closures need never be allocated.

More specifically, as implemented in the **LIL** backend, every recursive call (including all calls to other functions in the same nest) is implemented as a direct code call. In addition, every call to a function defined in the same lexical scope as the call is implemented as a direct code call. It is straightforward to extend this to functions defined in any enclosing scope: however this may increase closure sizes and hence is not done in the **LIL** backend.

In addition to calling code functions directly, some additional optimizations are performed on the calling sequences of non-escaping functions (that is, functions which are only called directly as code). For such functions, the environment record is never actually constructed. Instead, the components of the environment are passed as additional arguments to the function (just as with the varargs optimization for ordinary records). Floating point (64 bit) elements of the environment are further optimized by passing them unboxed on the stack. For all functions, environments with only a single element are "unwrapped": that is, the underlying singleton element is passed unboxed, instead of boxed in an environment. Finally, all closed closures are statically allocated as data (a closure may be closed because its environment is empty, or because all of its constituents are statically allocated data).

One of the advantages of using a type erasable language is that the treatment of types in closure conversion is much simpler than in the type passing case. Since types are used at compile time only, type environments can be passed immediately via application, instead of being packed into an existential. Implementing closure conversion for types was straightforward.

## 9.4 General Optimization

The process of encoding types as term data and making typecase explicit exposes new structure to the compiler which is available for optimization. Moreover, it greatly complicates other phases of the backend to be overly concerned with generating no dead, redundant, or otherwise sub-optimal code. Consequently, it seemed most expedient to port the **MIL** one pass optimizer to the **LIL** so that basic optimizations could be performed on **LIL** programs. The **LIL** optimizer performs CSE, dead code elimination, eta reductions for records and functions, projection from known records, reduction of known switches, folding of boxes and unboxes, constant folding, and reduction of coercions. Additionally, it makes some effort to recognize boolean terms which are used only as arguments to switches, and hence which can be compiled to produce jumps to branch targets directly from comparisons, instead of evaluating the term to produce a boolean value, and then re-branching on this result.

The general optimizer is most useful to help clean up after the closure converter, since closure conversion involves a fairly substantial rewriting of programs. Many box/unbox reductions become available after closure conversion, since floating point terms are currently boxed in environments. Numerous record projections and record formations can be eliminated as well in cases where only some or none of the environment is used in a given context. Multiple calls to the same function in the same scope can be optimized so that only one projection of the code pointer and the environment need be done.

All these optimizations (and more) are performed on the closure converted code. In general however, a great deal more could be done to optimize after closure conversion. In particular, I believe that the output of the closure converter could benefit greatly from some form of partial-redundancy elimination which could move code evaluated on only some paths down into the arms of switches. The closure converter introduces numerous projections from a function environment in the function header, some of which are only used conditionally in the function. It may also be forced to re-allocate closures for recursive escaping occurrences which are again only used conditionally in the function. Such optimization is beyond the scope of this thesis.

## 9.5 Lil To Tal

The implemented translation from **LIL** to **TALx86** is conceptually quite similar to the formal description given in chapter 8, despite the fact that the **TALx86** language is substantially different from the idealized **TILTAL** language. While details of the implementation of particular constructs are specialized to the **TALx86** type system, the overall code generation approach if very similar.

Code generation and register allocation are done simultaneously in (notionally) a single pass. In fact, for simplicity the code generator is run twice on each function: once using a virtual frame pointer and once using stack pointer based frames. The output of the first run is ignored: it is done only to determine the frame size so that offsets from the stack pointer can be computed directly. This was done for the sake of simplicity in the coding only: it would be straightforward to implement a second pass to eliminate uses of the frame pointer.

### 9.5.1 Code generation

As with the formal description, code generation is done by translating terms with a pre-assigned destination in a similar fashion to that described by Dybvig et al. [DHB90]. This is particularly convenient given the register allocation strategy chosen, in which let-bound variables generally have already been assigned a location when their bindings are translated. Consequently, operations which are bound to variables can often be translated in such as way as to compute their values directly into the appropriate location.

As in the formal development form chapter 8, terms are also translated with a context of occurrence which indicates the use to which the result of a computation is to be put. For example, terms which are translated in a "return" context are expected to de-allocate the frame and return directly, instead of returning a value. Consequently, general tail-call elimination falls out very naturally: function bodies that end in function calls (even when guarded by a conditional or a switch) simply de-allocate the frame and jump to the new callee. (Some additional adjustment of the stack is necessary when the callee has a different number of arguments than the caller).

Additional contexts of occurrence are used to generate efficient code for conditional branches, and for terms evaluated only for side-effects. A term whose value is used only to decide a branch is evaluated in a context of occurrence which provides appropriate continuations which can be called directly. A term whose value is unused is evaluated in a context of occurrence which indicates that no return value need be provided.

### 9.5.2 Register allocation

The assignment of variables and temporaries to registers and stack slots is performed simultaneously with code generation. Code generation is performed in a bottom up fashion, so that the live range of variables is apparent from the binding structure of a program: a variable becomes live when it is first encountered in the bottom-up pass, and becomes dead when its binding site is reached. Registers and stack slots are assigned to variables greedily, as needed by the generated code. When a variable use is encountered by the code generator, the register allocator is queried to provide the machine location in which the variable resides. The code generator dictates whether or not this location is permitted to be a stack slot, or must be a register.

#### Register assignment and spilling

The register allocator maintains state describing the assignment of variables to registers and stack slots and vice versa. This state encodes the assumptions which the previously emitted code makes about the state of the machine upon entry. When the register allocator is queried about a variable to which it has already assigned a machine location it returns the previously assigned location *if it is appropriate*. However, if the variable was previously assigned to a stack slot and the code generator requires that the use under consideration be assigned a register, the register allocator chooses a free register and changes the state to map the variable to this new location. Since previously emitted code assumes a different location for the variable, the register allocator is responsible for emitting code to initialize the previously assigned stack slot from the newly assigned register.

If a variable is required to be in a register and all registers are already assigned to variables, a variable must be chosen to spill to the stack. Since previously emitted code assumes that the variable was in a register, the allocator must emit code to load the register from the newly assigned stack slot. As a simple spill heuristic, the variable whose next use is farthest into the future (that is, whose next use is farthest forward in the instruction stream) is spilled. This is intended to attempt to provide more intervening instructions to hide the latency of the memory access, as well as to attempt to keep locally active variables in registers. No effort was put into attempting to validate this choice of heuristic as extensive tuning of the code generator is beyond the scope of this thesis. However, from informal inspection of the generated code, the allocator seems to usually make reasonable choices, though there are also cases where it could be improved.

If a location is requested for a previously unseen variable, it is assigned a register if possible, and otherwise a stack slot if permitted by the code generator. If neither of these cases apply, a variable must be spilled as described above to free up a register for the new variable.

At a variable binding site, the register allocator provides the code generator with the location into which the variable initialization code should write its value. Variables which have not been assigned a location when their binding site is encountered are unused, and the result of the initialization code may be safely discarded.

Unlike many register allocation schemes in which variables are assigned a single location with spill code emitted to load them into temporary registers when necessary, this register allocation approach allows variable/location assignments to be changed dynamically. Variables may be moved between stack slots and registers multiple times over the course of a function. In this sense this approach is similar to the "second chance binpacking" allocation strategy described by Traub, et al. [THS98].

#### Machine state types

An important side benefit of the register allocation data structures is that they provide, at any given point during code generation, a snapshot of the machine state. The register allocator maintains a mapping from variables to registers and frame locations. Consequently, it is straightforward in this framework to define the translation of a **LIL** context $\Gamma$ as described in chapter 8. The operation $|\Gamma|^{\sigma}_{\mathcal{A}}$ mapping a stack tail $\sigma$ and a **LIL** context $\Gamma$ to a machine type $\Gamma_A$ is trivial to implement given the data structures already used by the register allocator. The intention is that the register allocator as implemented should constitute a "good allocator" as described in chapter 8.

### 9.5.3 Additional TILT constructs

There are many more language constructs in the actual **LIL** language as implemented than in the formal **LIL** description. For the most part however, these are simply additional primitive operations on basic types, and are not especially interesting. However, two such constructs warrant additional comment.

#### Coercions

Vanderwaart et al. [VDP+03] describe a coercion calculus used in the **TILT** compiler to faithfully implement Standard ML datatypes in an efficient manner. Applications of coercions into and out of datatypes in this system incur zero runtime overhead. This system is quite simple, requiring only one new type constructor and three new term level constructs. However, it was not supported by the existing **TALx86** type system, and I preferred to modify this system as little as possible.

A simple technique for eliminating these coercions is to replace them with function calls. This approach has the benefit of placing no additional burden on the code generator and requiring no changes to the **TALx86** type system, but suffers from a substantial performance penalty [VDP+03]. As a compromise, I chose to implement coercions as functions with a highly specialized calling convention. In particular, they are implemented as functions whose argument and result are both in the same register, and for which all other registers are callee-save. Since the actual bodies of the functions do no runtime work, this means that the runtime cost of applying a coercion is reduced to two instructions: a call and an immediate return. There may be some additional cost since variables may need to be spilled or re-shuffled in order to place the argument value in the appropriate register before calling the coercion. However, in general the overhead is quite low.

#### External functions

In general, most programs need at some point to call external functions. At the very least, any interesting program must at some point make a system call to do input and output. These external functions are probably not in general provably safe, and are in any case not available in a verifiable form. Consequently, these external operations are generally considered part of the trusted computing base: that is, the safety of the checked code is certified relative to the correctness and safety of the external code. However, it is important that there be some mechanism for controlling which external code the certified program has access to, both so that the certification system cannot be trivially subverted, and so that the ability of mobile code to access system resources can be controlled by the host.

One approach to this is to explicitly incorporate the external functionality needed by programs into the runtime layer upon which the program runs. Essentially, the source and target language are extended with a statically known set of additional primitives. This has the benefit of making it easy to control the set of resources available to programs, and of routing access to these resources through a central authority (leaving open the possibility of doing additional sandbox style dynamic checks). A disadvantage of this approach is that it is generally difficult to extend the set of external functions available, since the runtime itself must be modified.

The approach taken in this thesis is to push the issue of access to external resources out to the linking phases. The **TILT** compiler extends the Standard ML language with a primitive function type classifying functions obeying the C calling convention, and a primitive application form for applying such functions. Any compilation unit may declare a set of external functions against which it is compiled. The generated **TALx86** modules list these functions as typed imports. The code in the compiled module is certified to be safe under the assumption that the imported labels are safe at their declared types.

The question of how these import assumptions are discharged then becomes a policy decision on the part of the host at link time. The **TALx86** linker provides three different modes by which import assumptions can be discharged.

The first is that a typed binary may be provided to satisfy the import assumptions. In this case, these assumptions will be explicitly checked by the linker, and so long as the **TALx86** type system is correct, the linked result is guaranteed to be safe.

The second mode is that an untyped binary can be provided, along with an export interface providing types for the exported labels. In this case, the **TALx86** linker can check that the import assumptions of the client module match the exported interface of the untyped binary, but cannot check that the untyped binary actually provides the given interface. Consequently, the safety of the linked program is contingent on the untyped binary actually matching its exported interface.

The third linking option is to simply link an untyped binary against the typed client code, with no interface checking whatsoever. In this case the safety of the linked program is contingent on the untyped binary matching the declared import assumption of the typed binary.

The difference between the second and the third case is important, but subtle. The key element is that in the third case, there are no restrictions on what functions the typed binary can call, and what types it assigns to them. Consequently, it is trivial to subvert the type system simply by providing a typed binary which imports functions at incorrect types. In the second case, this is not possible, since the host provides an explicit interface against which the typed client is to be linked. So long as the host provides the correct types for the exported functions (and so long as those functions are implemented correctly and safely), the client is unable to subvert the type system since it is constrained to respect the exported interface.

In the **TILT** typed backend, a combination of the first two approaches is used to implement the runtime layer upon which programs run. Some parts of the runtime functionality are implemented in

**TALx86** and hence can be checked directly. Other parts are implemented in C, with an explicitly provided interface. The runtime layer as currently implemented is very minimal and does not provide any support for system calls or any useful library functions. A version of the Standard ML Basis Library was modified slightly to provide this sort of operating system functionality. Currently, all of this code is linked using the third approach. This is satisfactory for using the compiler as a general x86 backend, but for use as part of a certifying compilation infrastructure it would be necessary to define a more limited interface to host resources, and to define explicit export interfaces against which client code could be checked.

## 9.6 Engineering

While extensive engineering of the **TILT** certifying backend is beyond the scope of this thesis, a certain amount of engineering work was required simply to demonstrate the feasibility of certified compilation in this setting. This section describes some of the techniques used.

### 9.6.1 Type representation

Unlike the **MIL**, the **LIL** lacks an internal definition mechanism for types. In the absence of any sharing mechanism whatsoever, types are observed to grow infeasibly large. The solution to this problem used in the **LIL** is to enable sharing via *hash-consing*.

Hash-consing is a popular technique for controlling representation size which works by looking up each newly allocated type into a table containing all previously allocated types. Whenever a previously allocated version is found in the table, the two copies may be be represented as pointers to the same data-structure. Hashing is used to make this reasonably efficient: each node in the parse tree of a type contains the hash value of the type, which can then be used to produce hash values for parent nodes. The table of allocated nodes can then be implemented as a hash table with reasonably fast lookup times.

Hash-consing ensures that all syntactically equal types can be shared. However, types which are alpha-variants of each other will not be shared in this scheme (except accidentally) since the hash codes of different variables will usually be different. This problem can be completely avoided by switching to a representation in which variables are represented via deBruijin indices, wherein a variable use is implemented as a count of the number of binding sites between the variable use and the variable binding site. When variables are implemented in this fashion, all types which are alpha-equal will also be syntactically equal, and hence can be shared via hash-consing.

However, deBruijin indices are notoriously difficult to work with, and give no guarantee of improved sharing. Moreover, all of the existing **TILT** infrastructure was built around using variables. Consequently, I chose to use a standard representation of variables. In practice, this seems to work well so long as care is taken to avoid introducing un-necessary alpha-variants.

In addition to sharing syntactically equal types in memory, it turns out to be very important to share kinds as well. In almost all programs, the number of syntactically distinct kind nodes is smaller than one-hundred. In the absence of hash-consing for kinds however, the number of actual kind-nodes in memory is commonly three or four orders of magnitude larger than this. In general throughout this section, all of the techniques discussed in the context of sharing types should be considered to apply to kind information as well.

### 9.6.2 Traversing programs

Sharing types and kinds in memory is crucial in order to reduce internal representations of programs to a manageable size: that is, to reduce the space usage of the compiler. However, it is almost equally important to ensure that in addition to avoiding redundant representation of types, the compiler also avoid redundant computation on types. Hash-consing permits type parse-trees to be represented more compactly as directed acyclic graphs (DAGs). Compiler passes that rewrite types represented in this form must also be careful to traverse these programs as DAGS.

Traversing programs in DAG form is implemented by providing an abstract lookup table in which hash-consed type nodes can be inserted. Node lookup tables are used in compiler passes to keep sets of nodes already traversed: or more generally, to keep mappings from previously traversed nodes to the traversal result (or other useful information). Before attempting to rewrite a node, the compiler pass checks the table for previously computed result. If no such result is found, the result is computed and entered into the table for future use. This allows passes to avoid re-traversing types unnecessarily without impacting the structure of algorithms in any significant way.

Compiler passes that produce different results based on contextual information are slightly more complicated to implement in this way, since it is no longer necessarily the case that syntactically equal types rewrite to the same results. This situation only really arose in the **LIL** typechecker which is discussed extensively below.

### 9.6.3 Fast substitution

A very common operation in the compiler is the substitution of types for free type variables in other types (as well as kinds for kind variables). In practice, substitution often proves to be an efficiency bottleneck in the **TILT** compiler. In order to make substitution faster, the **LIL** backend maintains sets containing free type and kind variables at each type and kind node. This allows substitutions for variables to only traverse those nodes which actually contain free occurrences of the substituted variables.

Another important benefit of maintaining sets of free variables is that it allows the substitution code to avoid introducing unnecessary alpha-variants of types. Capture avoiding substitution must in general alpha-vary when crossing a type variable binding, since the bound variable may occur free in the type being substituted. This alpha-variation may cause an entirely new copy of the type to be created, even if the substitution only affects small sub-trees of the type. This can be avoided in the case where the bound variable does not occur free in the type being substituted. Maintaining sets of free variables makes this check efficient.

### 9.6.4 Fast alpha-equivalence

The hash-consed implementation allows for a fast syntactic equality check, since any two syntactically equal types will be represented by the same type node and hence can be checked for equality in constant time. In cases where the equality check fails, a normal equivalence check must be performed, since alpha-variants of the same type will not in general be represented by the same node.

### 9.6.5 Weak Head-Normalization

Weak head-normalization is a common operation on types, since it is the first step in checking type equality. In addition to representing types as a directed acyclic graph, it is important to normalize them as graphs as well to avoid repeated reductions of the same terms. This is implemented in the **LIL** back end by associating with every hash-consed type node an optional weak head-normal form. The reduction algorithm then checks for a previously computed normal form before reducing the term and avoids re-computing it if present. If the normal form is not present it must be computed: but the new normal form can be stored with the node for use by subsequent reductions of the same shared node. This technique (among others) has been previously described and empirically characterized by Shao et al [SLM98].

### 9.6.6 Engineering the LIL type checker

A valuable benefit of using a typed internal language in a compiler is that it allows the output of each compiler phase to be checked to ensure that nothing untoward has happened. Many, if not most compiler bugs will cause a compiler to generate type incorrect code at some point. An internal type checker therefore provides the valuable service of catching and pinpointing compiler bugs before runtime, when they are likely to be much harder to track down.

In order to reap these benefits however, the compiler's internal type checker must be efficient enough to quickly check anything the compiler produces. However, the simple technique described in the previous section for traversing types as directed acyclic graphs cannot be applied naively to a type checker. The reason is that type checking is a context sensitive operation: syntactically equal types may be well-formed in one typing context and not in another (for example because of the presence of free variables referring to different binding sites). In the **TILT** compiler, maintaining the simplicity of the typechecker implementation was an important goal since the correctness of the typechecker is both important and hard to test (at least in terms of soundness). However, the straightforward approach of completely ignoring sharing of types and kinds proved completely impractical. Typechecking small programs easily exhausted the resources of a Pentium 4 machine with one gigabyte of RAM. As a first step, it was necessary to find a way to avoid redundant type checking.

#### Exploiting physical sharing in the type checker

While the nature of typechecking is context sensitive, it is by far the most common case that checking syntactically equal terms will produce the same result (that is, that the terms will be well-typed at the same kind). Moreover, it is most frequently the case that two syntactically equal terms are judged to be well-typed for the same reasons: that is that the contexts in which they are checked are (essentially) equal. A first cut at improving the performance of the typechecker then is to keep a table mapping types to contexts in which they have already been judged equivalent (as well as the kind classifying them). Before checking a type, the compiler first checks a lookup table to see if the type under consideration has been checked before in an equivalent context (and with an equivalent kind).

Strict pointwise equality on typing contexts could in principle be somewhat expensive to compute, and moreover is vastly more restrictive than is necessary. It is frequently the case that two syntactically equal terms will be checked in two contexts which differ, but only in irrelevant ways (for example, one context may be a well-formed extension of the other). An improvement on the simple technique described in the previous paragraph is to observe that so long as the two context are both well-formed and are pointwise equal on the free variables of the type under consideration, it is unnecessary to re-check the term in the new context.

These techniques suffice to permit traversal of types as DAGs in the **LIL** typechecker with a fairly small overhead.

### Context substitutions

The technique described in the previous section avoids most redundant checking of types, but is not sufficient in and of itself to make checking large programs practical in **TILT**. The **LIL** presents an unusual engineering challenge because of the nature of its type analysis mechanism. Several of the **LIL** type checking rules induce substitutions into typing contexts as part of the type refinement mechanism. For example, in one of the rules for type checking type pair refinements, an explicit type pair is substituted for a variable in the typing context.

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

This context substitution operation ($\Gamma[\langle\beta,\gamma\rangle/\alpha]$) turns out to be quite expensive, since implemented naively it involves duplicating the entire typing context after every type refinement. The fast substitution cut off afforded by the free variable sets on type nodes means that types in the context are not traversed unnecessarily, but in the absence of a very specialized data structure, the container implementing the context must be copied in entirety. Since type refinement operations are quite frequent in many programs, this rapidly becomes unmanageable.

There are numerous approaches that one could take to making this more efficient. The approach taken in the **LIL** backend is based primarily on two observations. Firstly, many elements of a typing context will not be affected by the context substitutions at all. Of those that are, many of them will not actually be referenced in any given term: it is often the case that the body of a type refinement construct will have at most two or three free variables. Any effort spent traversing unreferenced context elements is wasted.

Secondly, it is common to encounter several type refinement operations consecutively. Implemented naively, this means that multiple passes over the typing context may be made with no intervening variable lookups.

The **LIL** implementation of context substitution therefore attempts to do two things. In order to take advantage of the second property, substitutions are aggregated and carried out lazily. Substituting into a context simply records the substitution without carrying it out. Multiple substitutions can consequently be aggregated together. In order to take advantage of the first property, these aggregated substitutions are then only carried out on the result of a variable lookup. Full context substitutions are never performed.

There are a number of subtle points involved in this optimization. It is clearly not valid to simply maintain a substitution mapping variables to types and apply it to every type that is looked up in the context, since different substitutions will have been in place at different binding sites. In the next section, I give an informal presentation of an extension to the static semantics of the **LIL** that captures this optimization, and then briefly discuss the differences with the actual implementation.

**Fast context substitution**

As a first cut, I extend the notion of a typing context $\Gamma$ with substitution nodes. For brevity, I elide 64 bit context entries - their addition is trivial.

$$\Gamma ::= \bullet \mid \Gamma, x{:}\tau \mid \alpha \to c$$

In order to ensure that substitutions are carried out correctly, I replace the implicit use of context re-ordering in the **LIL** static semantics with explicit re-ordering rules that carry out substitution nodes as needed.

$$\frac{\Psi; \Delta; \Gamma, y{:}\tau_y, x{:}\tau_x, \Gamma' \vdash sv : \tau}{\Psi; \Delta; \Gamma, x{:}\tau_x, y{:}\tau_y, \Gamma' \vdash sv : \tau} \qquad \frac{\Psi; \Delta; \Gamma, \alpha \to c, x{:}\tau_x[c/\alpha], \Gamma' \vdash sv : \tau}{\Psi; \Delta; \Gamma, x{:}\tau_x, \alpha \to c, \Gamma' \vdash sv : \tau}$$

The variable lookup rule is then restricted in the usual way, requiring that variables be re-ordered to the end of the context for lookup.

$$\frac{\Delta \vdash \Gamma, x{:}\tau \ \mathbf{ok} \quad \vdash \Psi \ \mathbf{ok}}{\Psi; \Delta; \Gamma, x{:}\tau \vdash x : \tau}$$

Context substitutions can then be implemented in terms of these substitution nodes.

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

This version of the variable pair refinement rule simply adds a substitution node $(\Gamma, \alpha \to \langle \beta, \gamma \rangle)$ instead of explicitly substituting for $\alpha$ as in the original version $(\Gamma[\langle \beta, \gamma \rangle / \alpha])$.

This extension to typing contexts captures the notion that variable substitution need only be performed on referenced variables, and can be deferred until the point of reference. This easily extends to capture the idea of aggregating substitutions by replacing the single substitutions in the context with simultaneous substitutions for multiple variables. A simultaneous substitution $\Theta$ maps variables to types.

$$\Theta ::= \bullet \mid \alpha \to c, \Theta$$

The operation of a substitution $\Theta$ on a type $c$ is defined by a function $c[\Theta]$ mapping types to types.

$$\begin{array}{lll} c[\bullet] & \stackrel{\text{def}}{=} & c \\ \alpha[\alpha \to c, \Theta] & \stackrel{\text{def}}{=} & c \\ \alpha'[\alpha \to c, \Theta] & \stackrel{\text{def}}{=} & \alpha'[\Theta] \quad (\alpha \neq \alpha') \\ \ldots & \stackrel{\text{def}}{=} & \ldots \end{array}$$

The remaining cases proceed compositionally over the structure of types exactly as with a normal substitution (taking care to avoid capture when crossing binding sites). Composition of simultaneous substitutions $\Theta \circ \Theta'$ is defined explicitly as an operation on substitutions obeying the equation $c[\Theta \circ \Theta'] = (c[\Theta])[\Theta']$.

$$\begin{array}{lll} \bullet \circ \Theta & \stackrel{\text{def}}{=} & \Theta \\ (\alpha \to c, \Theta) \circ \Theta' & \stackrel{\text{def}}{=} & \alpha \to (c[\Theta']), (\Theta \circ \Theta') \end{array}$$

Aggregation of substitutions in typing contexts is implemented by replacing single substitution nodes with simultaneous substitution nodes.

$$\Gamma ::= \bullet \mid \Gamma, x{:}\tau \mid \Gamma, \Theta$$

The re-ordering rule for substitution nodes is modified appropriately.

$$\frac{\Psi; \Delta; \Gamma, \Theta, x{:}(\tau_x[\Theta]), \Gamma' \vdash sv : \tau}{\Psi; \Delta; \Gamma, x{:}\tau_x, \Theta, \Gamma' \vdash sv : \tau}$$

A new structural rule is introduced permitting adjacent substitutions to be aggregated using substitution composition.

$$\frac{\Psi; \Delta; \Gamma, \Theta_1 \circ \Theta_2, \Gamma' \vdash sv : \tau}{\Psi; \Delta; \Gamma, \Theta_1, \Theta_2, \Gamma' \vdash sv : \tau}$$

### Fast context substitution in practice

The actual implementation used in the **LIL** backend differs from this presentation slightly. While the formal system implements typing contexts as lists, the implementation uses a splay tree implementation for efficient variable lookup. In order to avoid the need to design and implement a custom balanced tree implementation supporting explicit substitution nodes, I chose instead to maintain the substitution nodes in an auxiliary data-structure, allowing the core splay tree data structure to remain unchanged.

Notionally, the implemented version can be derived from the system described above in the following manner. Number the substitution nodes in a typing context, starting from the "leftmost" substitution node. With every variable/type pair in the context, associate the number of the first substitution to its right (that is, the first substitution that applies to it). Finally, remove the substitution nodes into a separate data structure associating each substitution with its index.

Upon looking up a variable, apply the composition of all substitutions with indices greater than or equal to that associated with the variable (that is, all substitutions that were to the right of the variable in the original context). This corresponds to the sequence of substitution that would have been performed by the sequence of structural re-ordering steps required to move the variable to the end of the context. As an optimization, this composition of substitutions may be computed eagerly.

Upon inserting a variable into the context, associate with it the first index larger than the largest currently used index.

Upon substituting into the context, associate the new substitution with the first index larger than the largest currently used index. The composition of this substitution with previous substitutions may be eagerly computed if it is more efficient to do so. If no variables have been inserted into the context since the last context substitution was performed, it is also possible to simply replace the substitution associated with the largest index with the composition of the old substitution and the new substitution, avoiding the addition of a new substitution node. Intuitively, this corresponds to eagerly applying the context composition structural rule.

This algorithm provides a straightforward implementation of explicit substitution nodes in a typing context without requiring any modification to the underlying data structure. There are additional improvements possible to the algorithm. For example, in the static semantics as presented above, it is possible to (non-deterministically) choose to apply the context re-ordering rules for a variable at the first point in a derivation past which more than one immediate sub-derivation contains a lookup of that variable. In this way, the application of any given substitution to the type associated with a variable can be guaranteed to be performed at most once. The algorithm as implemented does not implement this: the substitution is re-applied to the original type upon every variable look up. In practice this does not seem to be a problem. However, if after further engineering this became a bottleneck, it would be straightforward to memoize the application of substitutions to ensure that no unnecessary traversals of types is done.

## 9.7 Compilation units

The definition of Standard ML [MTHM97] does not define a notion of separate compilation. In order to remedy this, the **TILT** compiler defines a notion of interface that generalizes Standard ML signatures, and uses these interfaces to mediate between separately compiled source units. Any unit of source code may be compiled in complete isolation from any other unit upon which it relies, so long as suitable interfaces are provided.

The certifying backend described in this dissertation implements the full **TILT** separate compilation system. This is done by viewing each compilation unit as a functor which maps its imported units to its exports. In this way, the linking process is modeled as function application, avoiding the need for a complicated type theory to track the initialization of globally visible locations.

More concretely, a **LIL** compilation unit may be thought of as a pair consisting of a type component and a term component. The type component is a type function mapping imported types to exported types. Similarly, the term component is a function mapping imported terms to exported terms. In actuality, the type portion is implemented not as a function, but instead by using the **TALx86** linker directly: each unit lists imported types with their kinds as explicit imports instead of parameterizing the components over them.

A **LIL** interface classifying such a compilation unit is a translucent sum: the first component of which is a kind classifying the type element of the compilation unit, and the second component of which is a type classifying the term element of the compilation unit. The pair is dependent in the usual manner: the term classifier portion may refer to the label of the type classifier portion.

The **TALx86** linker as defined and implemented by Glew et al. [GM99] does not in fact support translucent sums, and so it was necessary to extend the **TALx86** implementation with an alternative (and simpler) notion of typed linking which implements the standard translucent sum matching rules. This was the only significant change needed in the **TALx86** infrastructure.

### 9.7.1 Compiler generated files

The **LIL** backend emits four files for every compilation unit. The first is a typed assembly file named **asm.tal** containing the decorated assembly code for the unit: this corresponds to a **LIL** compilation unit as described above. The interface of a compilation unit (whether explicit or derived) is compiled to a term export file named **asm_e.tali** classifying the exported term portion of the unit, and a type export file named **tali** classifying the exported type portion. Finally, an additional file is emitted containing the signatures of any external C functions used by the unit.

The **TALx86** assembler type checks and assembles the typed assembly file (**asm.tal**) to produce two object files: a standard object file (**obj.o**) and a type object file (**obj.to**) containing the type annotations necessary for typechecking the object file. In the process, it verifies that the provided code matches the interfaces specified in the **asm_e.tali** and **tali** files.

After compiling each individual unit in a program, **TILT** must also produce an additional unit to serve as the "link" unit for the entire program: that is, a unit which applies each compilation unit function to its imports to produce its exported result. This exported result is then passed on to subsequent units which import it. This "link" unit implements the ML level linking. The **TALx86** linker is used to typecheck and construct the final executable program, linking together the "link" unit with each of the compilation units providing the individual unit initialization functions.

## 9.8 Measurements

The goal of this dissertation is to demonstrate the feasibility of certifying compilation in a type analysis framework. The bulk of the dissertation is concerned with simply developing and describing the framework necessary for this process. This provides the first argument for feasibility: that it is possible at all. This section is intended to provide some evidence that not only is it possible, but in fact that it is practical. In the following sections, I present some empirical measurements of the performance of certifying **TILT**. In particular, I present measurements quantifying the size of the generated type annotations on the emitted code, and measurements of the run time of various benchmark programs with comparisons to other compilers.

It is important to re-iterate here that engineering the compiler to improve its behavior along either of these axes is beyond the scope of this thesis. In both cases, the engineering goal was simply to develop a working system, and to measure the result. No significant effort was spent on improving the system based on these measurements, and inspection of the emitted code suggests that substantial improvements along both of these axes could be implemented without running into any fundamental limitations of the framework.

### 9.8.1 Benchmarks

The benchmark programs measured in the following sections were selected to be representative of a number of different sorts of programs, ranging in size from 24 to more than 2000 lines of code. Each benchmark was compiled separately to an object file, and then subsequently linked into a testing harness from which it was run. The benchmarks were also linked against the full Standard ML Basis Library which provides many of the basic data types for Standard ML, along with access to system I/O facilities. Several of the benchmarks use additional data structure implementations from the SML/NJ library. Figure 9.3 lists the benchmark programs used in this section, along with their sizes (in lines of code). Note that the size given is for each single benchmark file only: code from other compilation units (such as libraries and the test harness) are not included in this count.

### 9.8.2 Type size

As discussed in chapter 1.4, one of the most commonly cited applications for certifying compilation is as a security mechanism for mobile code. Certified code that is downloaded to be run from potentially un-trusted sources (or over un-trusted communication channels) can be checked for any violations of the safety properties implied by the type system. An important property of a such

| Benchmark name | Description | Lines |
|---|---|---|
| Taku | uncurried function calls | 24 |
| Takc | curried function calls | 25 |
| Fib | fib, fact with default Int | 38 |
| Fib32 | fib, fact with 32 bit ints | 39 |
| PI | approximation of pi (fp) | 28 |
| Msort | Merge sort (lists) | 48 |
| ISort | Insertion sort (lists) | 50 |
| Quick | Quicksort (version 1) | 130 |
| Quick2 | Quicksort (version 2) | 152 |
| Life | Game of life (lists) | 205 |
| FFT | Fast fourier transform | 271 |
| PQueens | P queens problem (arrays) | 292 |
| Frank | Small theorem prover | 473 |
| Leroy | Knuth bendix completion (exceptions) | 537 |
| BarnesHut | N-body simulation | 684 |
| Simple | Spherical fluid dynamics | 860 |
| Tyan | Grobner basis calculation | 896 |
| Boyer | Theorem proving | 959 |
| Lexgen | Lexical-analyzer generator | 1178 |
| Pia | Perspective inversion algorithm | 2074 |

**Figure 9.3:** Benchmarks

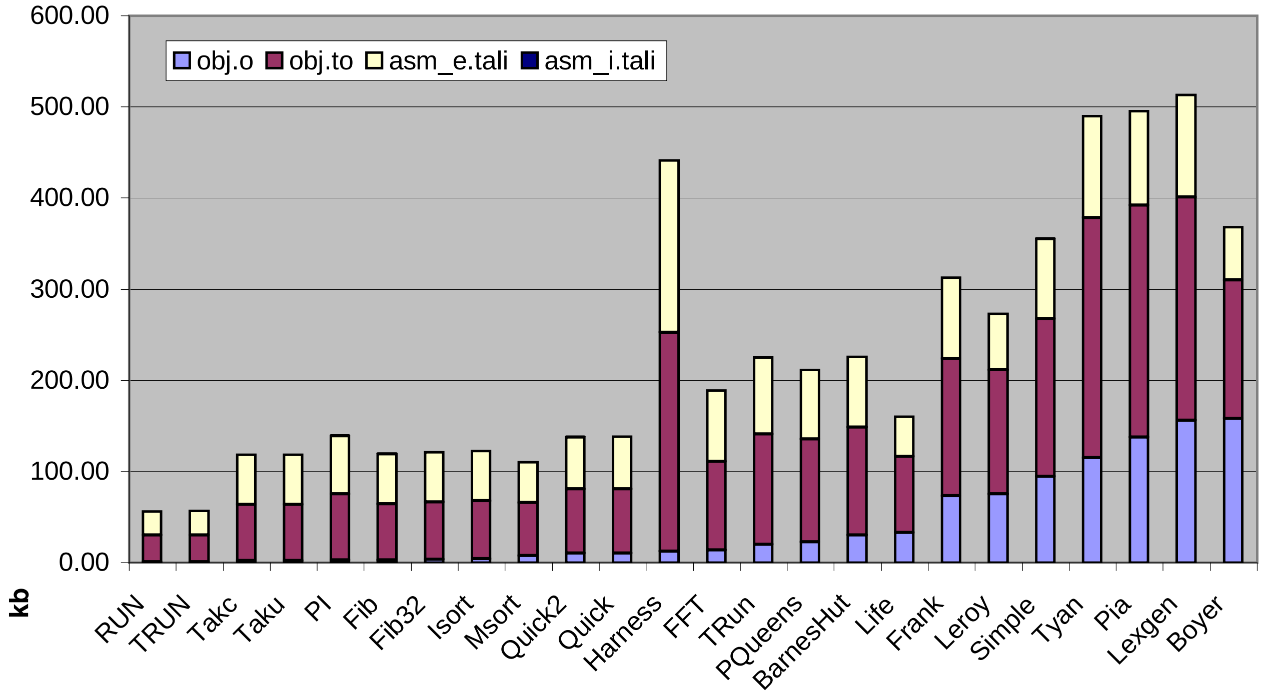



**Figure 9.4:** Size breakdown of individual benchmark units, in kilobytes.

a certification system is that the certificate size should not be unduly large so that the additional security provided does not come at a prohibitive cost in terms of bandwidth needed. In this section, I provide measurements quantifying the size overhead of the certificates generated by **TILT**.

In section 9.7.1 I described the files generated by the **TILT** compiler and the **TALx86** assembler. At the assembly code level, there are notionally two elements of a **TILT** compiled binary: a typed assembly file containing the actual assembly code annotated with type information (**asm.tal**), and some additional files describing the type of the exported interface provided by the binary (the **asm_e.tali**, **asm_i.tali**, and **tali** files). The typed assembly file refers to the exported interface files of any units which it uses, and any units which use it will in turn refer to its exported interface files. Interface files mediate between compilation units, while the type annotations within an assembly file allow individual units to be checked.

The typed assembly file is further split by the assembler into a standard untyped object file (**obj.o**) and a type annotation file (**obj.to**) that contains sufficient information to reconstruct the annotations on the untyped object file.

A reasonable measurement of the certificate size for a compilation unit after assembly then is the sum of the sizes of its export interface files (the **asm_e.tali** , **asm_i.tali**, and **tali** files) and the type annotation file generated by the assembler (**obj.to**), since these represent the incremental contribution of each compilation unit to the overall certificate size of the entire compilation system. Note that all of the interface information is present only to support separate compilation. Once linked together, almost all of the interface information can be discarded: in this sense this measurement is an upper bound.

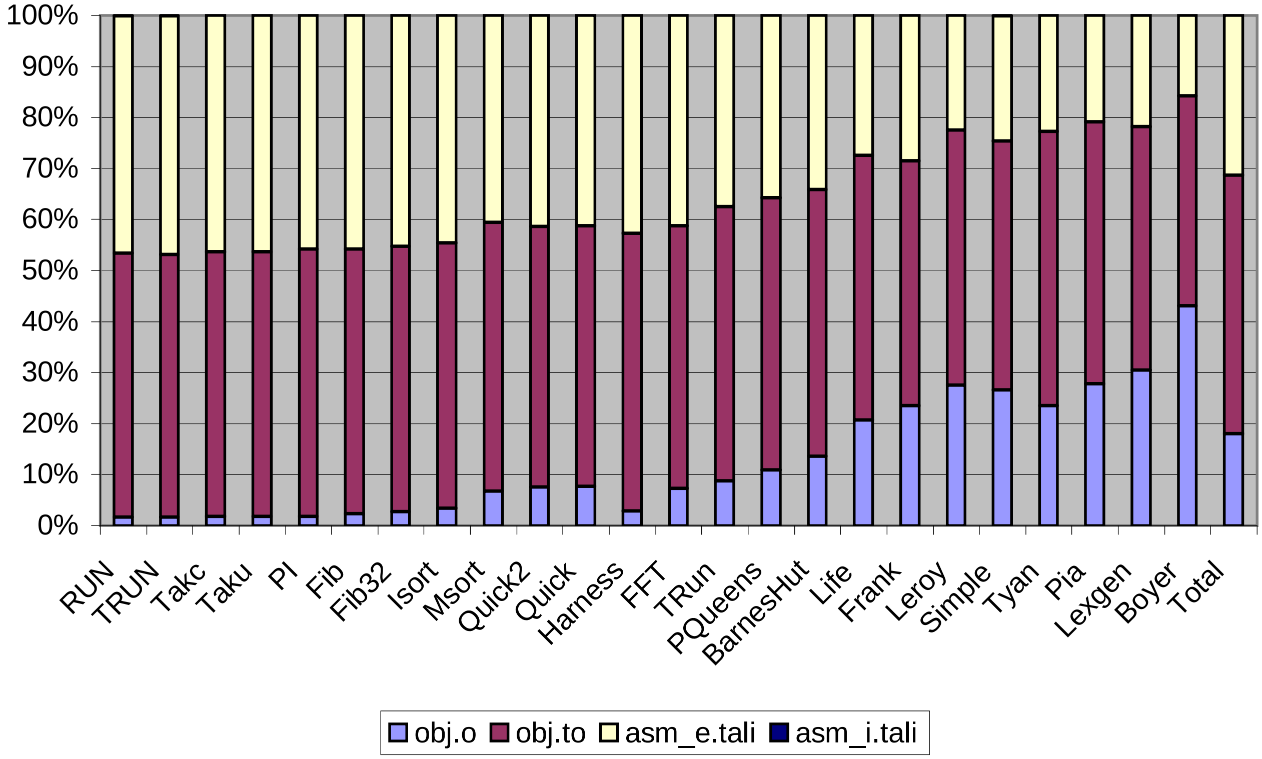



**Figure 9.5:** Size breakdown of individual benchmark units, relative.

### 9.8.3 Benchmark unit sizes

In this section, I give some size measurements for the **TILT** benchmark suite. Each of the units under consideration is a single compilation unit compiled from a single source file to a single object file. These object files are linked together with libraries, a test harness and the link unit to produce the final executable. Note that only the benchmark units themselves are included in this section. Measurements of the libraries and the link unit are discussed in the next section.

Figure 9.4 gives the absolute size in kilobytes of each benchmark. The columns are sorted by increasing size of the generated (untyped) object file. The segments of each column indicate the contribution of each of the different files making up the compilation unit. The size overhead of the safety certificate is everything other than the object file itself (the bottom segment). Note that in all cases, the contribution of the **asm_i.tali** file (containing the types of imported C functions) is too small to be visible.

Figure 9.5 provides a different view of this same data: in this figure each segment of each column indicates a percentage of the total size of the unit contributed by a particular source. For very small programs, the amount of type information dominates the object code size. For larger programs, the percentage of the total space usage devoted to the certificate decreases. This reflects a certain amount of fixed overhead required for each program: basic types such as exception handlers, types for printing routines, etc. As programs get larger, the cost of this fixed overhead goes down. For the largest of the benchmarks, the certificate occupies roughly sixty percent of the total size. Summed over all of the benchmarks, the certificate information occupies roughly eighty percent of the total space, indicating roughly a factor of five space penalty for certification.

**Export interfaces**

The topmost visible segment in both graphs represents the contribution to the total size of the exported interface of each of the object files. Generally speaking, this component is of comparable size to the type information in the object file itself. As would be expected, for bigger programs the percentage overhead of the exported interface decreases somewhat, since the overhead is amortized over a larger amount of actual object code. In fact, this overhead is probably almost entirely eliminable, for three reasons.

First, by the nature of the compilation approach taken in the **LIL** backend, each of these export interfaces describes a single function mapping each of the object files logical imports to its logical exports. Consequently, the size of this file increases significantly with the number of imported units. In addition, this implies that almost all of the type decorations in an export interface file are already present in the export interface files of its ancestors. It is very likely that this redundant information could be eliminated entirely by exporting a single canonical abbreviation for the exported type of each unit along with its exported type component.

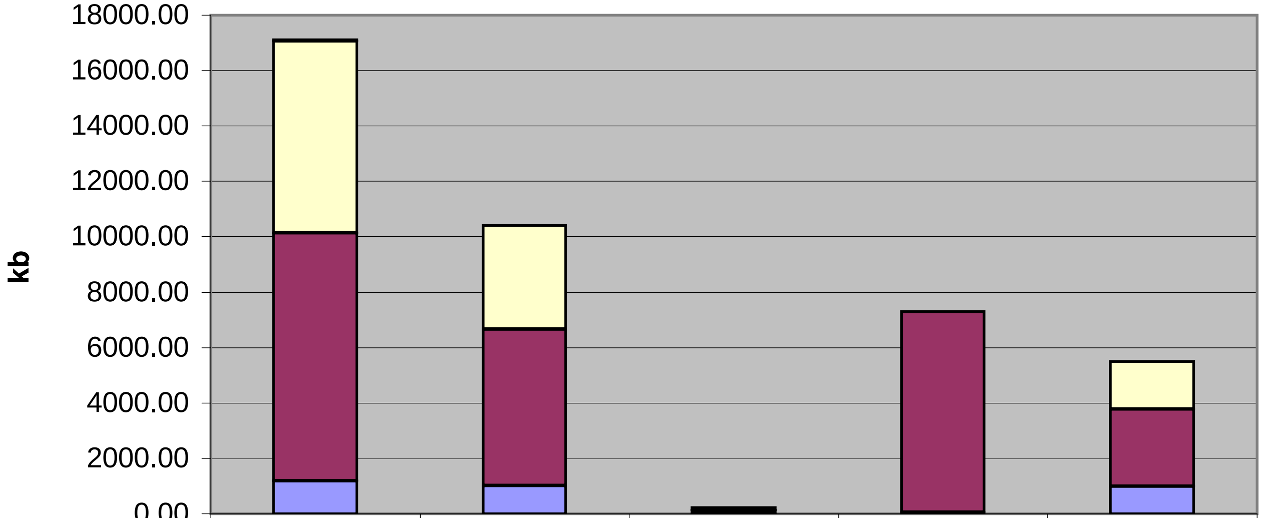
Second, because **TILT** implements separate compilation, the interface file and the actual implementation files may be produced and used independently. As a consequence, each contains its own copy of the type of the main exported initialization function (in fact, the entire export interface file consists solely of this). It is clear that in the common case where both files are produced as part of the same compilation, this redundancy could be factored out into a common definition site. This is almost certainly expressible in the **TALx86** linking system without modification.

Finally, note that these export interfaces exist solely to mediate between compilation units. Linked as a whole program, all of this interface disappears. Moreover, in many cases (such as in the benchmark suite), almost none of the entry points described by these interfaces are exported from the local group of compilation units. Even in the Standard ML Basis Library, there are many compilation units whose logical scope is entirely local to the library. A partial linking strategy wherein groups of compilation units are closed up into a single file presenting a single export interface would almost certainly eliminate a great deal of this overhead in most cases, even when whole program linking is not possible.

### 9.8.4 Libraries and linking

The measurements presented in figures 9.4 and 9.5 cover only the benchmark programs and the testing harness. These files must be linked against additional libraries before running: the Standard ML Basis Library, the SML/NJ Library, and a small command line argument parsing library. It is likely that in a certified compilation system, a subset of these libraries would be provided by the client, rather than as part of the certified binary. This is particularly true for the Standard ML Basis Library, which encapsulates the operating system functionality. Nonetheless, it is useful to measure the behavior of the compiler on these libraries as well as part of the overall system.

In addition to the libraries, there is one additional compilation unit that makes up part of the final executable program. As discussed previously, the compilation strategy employed in **TILT** maps each compilation unit to a single entry point implementing a function which takes as arguments the logical imports of the unit, and computes the logical exports as a result. The final step in compilation then is to create a unit which stitches together the whole program at runtime by applying each of these functions in turn. I refer to this unit as the link unit, since it implements the logical linking of the program. Note that this is distinct from the **TALx86** notion of linking,




| | Basis | Lib | Arg | Link | Bench |
|---|---|---|---|---|---|
| asm_i.tali | 50.17 | 4.45 | 0.11 | 0.00 | 1.96 |
| asm_e.tali | 6917.00 | 3731.66 | 91.22 | 0.00 | 1718.69 |
| obj.to | 8953.31 | 5658.56 | 117.81 | 7264.31 | 2787.91 |
| obj.o | 1183.08 | 1005.05 | 11.09 | 46.05 | 988.47 |

**Figure 9.6:** Total sizes of compilation groups (kilobytes).

which is used to resolve the free references within the link unit back to the exported entry points of each compilation unit.

Figure 9.6 compares the absolute sizes in kilobytes of each of the major compilation groups of the benchmark suite: the three libraries (the Basis, the SML/NJ Library, and the command line argument library), the link unit, and the benchmarks themselves (including the testing harness). As before, each column is broken into segments showing the contribution to the total of each of the constituent files. Also as before, figure 9.7 presents the same data as a percentage of the total for each group.

**The Basis Library**

There are several interesting points to note here. Firstly, it is clear that the Basis library (and to a lesser extent the SML/NJ library) have more certificate overhead than the benchmark suite. (This is also true of the argument library, but this is most likely because of its very small size.) While the Basis and the benchmarks viewed as a whole have similar amounts of actual object data, the certificate size for the Basis is larger by roughly a factor of four. While I have not investigated this phenomenon in detail, I conjecture that it is likely that this is partially a result of the particular architecture of the Basis library. Within the Basis, there are numerous units which consist solely of a few (or even one) functor applications, or which contain structures which aggregate numerous other structures together as sub-structures. Such units produce almost no object code, but have quite large types.

Further evidence for this can be seen in figure 9.8, which presents the contributions of each of

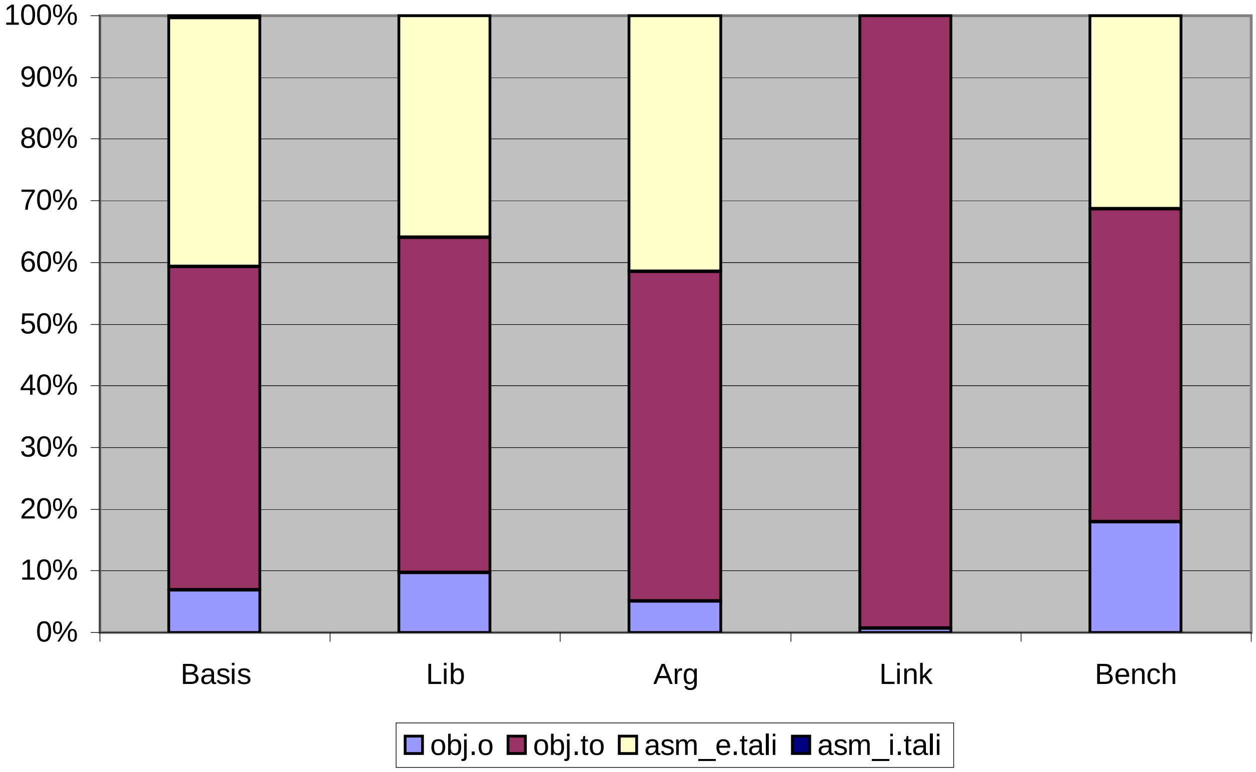



**Figure 9.7:** Total sizes of compilation groups (relative).

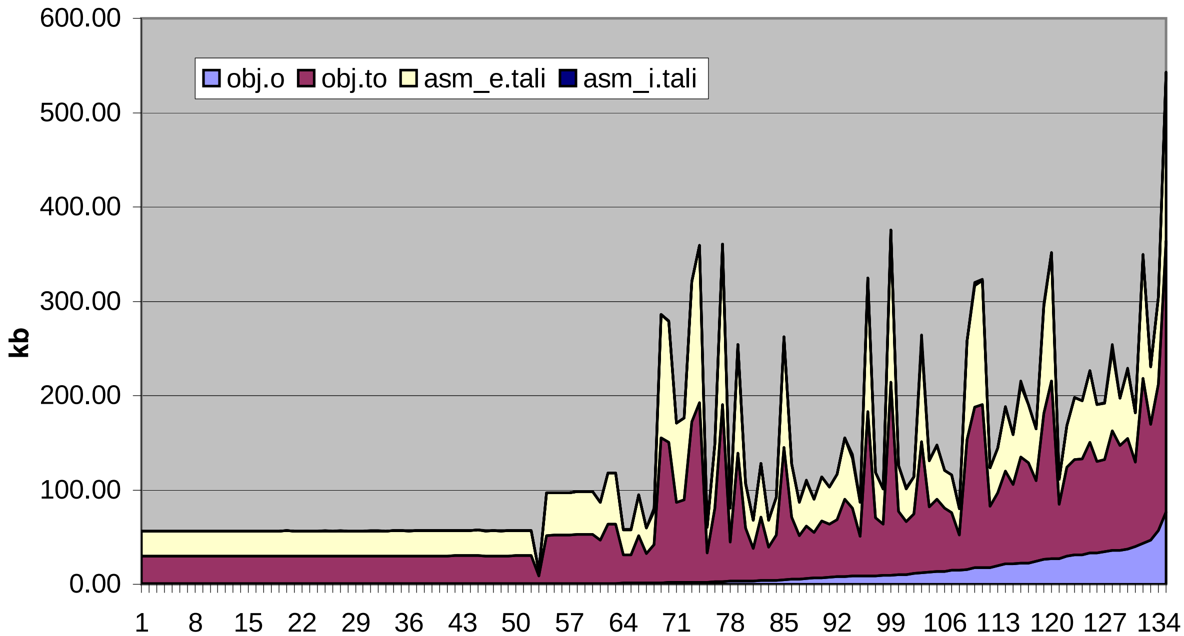



**Figure 9.8:** Breakdown of certificate overhead across individual Basis units (kilobytes)

the constituent elements for each compilation unit in the Basis library. For clarity given the large number of files, the graph is presented as a continuous area chart. As with the previous graphs, the units are sorted by object file size. For the most part, the certificate size (the sum of the top components) increases fairly smoothly with object file size. However, there are a number of very substantial spikes at points in the graph where very small files generate certificates comparable to those of the largest files in the library. All of these spikes that I have examined in detail correspond to the sort of aggregating files discussed above. Since for such files, much of the type information will have already been written out in the export files of the logically imported compilation units, it is likely that much of this overhead could be eliminated, even in a separate compilation setting (since the type decorations needed to describe the logical imports must be present in the interface file, which is in turn needed for separate compilation). However, in the current framework these small units with large types increase the aggregate certificate overhead substantially.

### The link unit

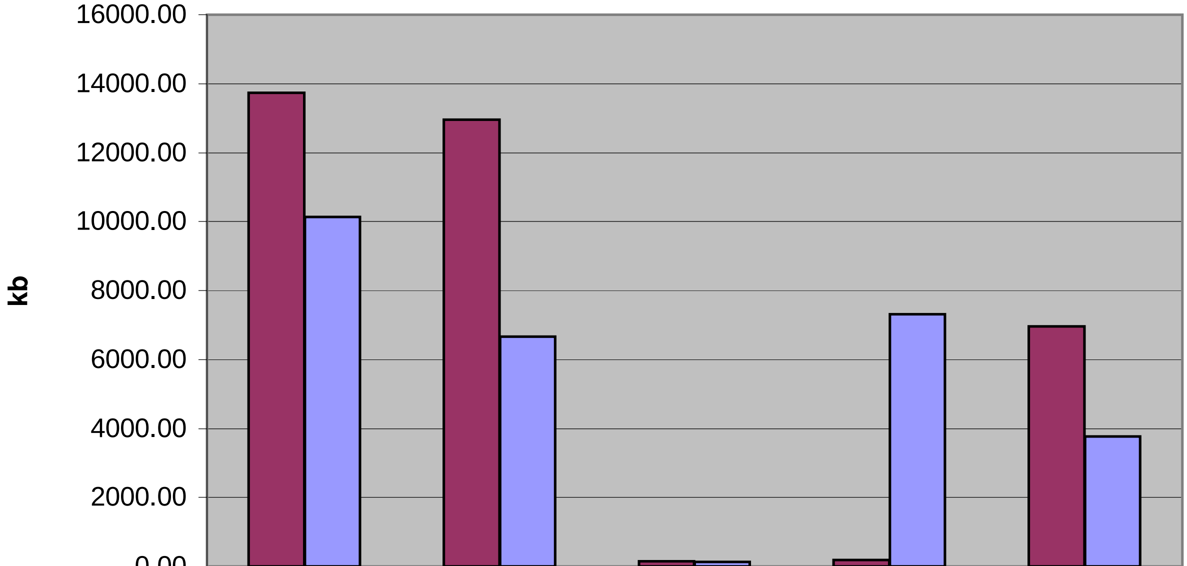
A second interesting observation about figures 9.6 and 9.7 is that the size of the link unit is entirely dominated by the certificate size. The actual object code for the link unit makes up less than one percent of the total size, and the total size itself is substantial in absolute terms (significantly larger than all of the benchmarks put together). In principle this is actually somewhat understandable. The link unit refers to every compilation unit in the entire program, including all of the libraries. It therefore must be able to refer to the type of the exported entry point (and its result) for each compilation unit. In some sense then, it is not surprising that the size of the certificate for the link unit should be comparable to the sum of the sizes of the export interfaces for all of the units in the program.

However, all of the type information needed to describe the entry points of the compilation units must be present in the export interfaces of the compilation units themselves. And since these export interfaces are needed by the link unit, there is no reason that the link unit needs to contain its own copy of these types. In fact, the typed assembly code produced by the **LIL** backend takes advantage of this property by using the abbreviation mechanism provided by the **TALx86** system. The link unit code generated by the **LIL** backend simply refers to the type of a compilation unit by using a canonical abbreviation name that is given a definition by the export interface of the unit. The assembly code produced by the **LIL** backend for the link unit is consequently quite compact. Surprisingly, the **TALx86** assembler seems unable to preserve this compact representation.

Figure 9.9 compares the aggregate sizes of the compilation groups before and after assembly. For each compilation group, the first column represents the total size of the assembly (**asm.tal**) files for the unit, and the second column represents the sum of the object and typed object files (**obj.o** and **obj.to**). Note that the files from these two columns represent the same information. The object and type object files (**obj.o** and **obj.to**) are produced by assembly the assembly file (**asm.tal**), and can be subsequently dis-assembled to re-produce the original assembly representation.

In general, the **TALx86** assembler does a very good job of managing certificate sizes. In all cases except the link unit, the assembled version is noticeably smaller than the original assembly files. This is universally true for individual files within the compilation groups, as well as in aggregate. The only exception to this is the link unit: in this case the assembled version is almost forty times the size of the original assembly file (even though notionally they represent the same information).

While I have not investigated this in detail, I believe that this is almost certainly because of a failure of the **TALx86** assembler to preserve the sharing from the original assembly file. I




| | Basis | Lib | Arg | Link | Bench |
|---|---|---|---|---|---|
| asm.tal | 13740.51 | 12948.89 | 155.55 | 188.54 | 6963.54 |
| obj.o+obj.to | 10136.38 | 6663.61 | 128.90 | 7310.36 | 3776.37 |

**Figure 9.9:** Assembly file size vs assembled output size (kilobytes)

conjecture that the references to abbreviated types from imported units in the original assembly file are being expanded out in the type object file. I do not believe that there is anything inherent to the structure of the link unit that prevents this sharing from being maintained: it should be possible to engineer an assembler to preserve this information.

### Scalability

The data in figure 9.5 suggest strongly that the certificate overhead scales well as the size of compilation units increase. For larger units, the overall percentage overhead is substantially smaller than the overhead for smaller units. In order to demonstrate the scalability of the system further, I took additional measurements on two very large programs.

The first of these large programs consisted of a subset of the benchmarks along with all of the library code upon which they rely, concatenated into a single file. In addition to the SML/NJ libraries, this included a large portion of the Standard Basis Library as well. Unfortunately, because of a limitation with the **TILT** foreign function interface, it was not possible to concatenate the entire Standard Basis Library into a single valid Standard ML unit. Therefore, certain of the benchmarks (such as those performing file i/o) could not be included. The resulting file consisted of 8332 lines of code.

The second of these large programs consisted of approximately half of the **TILT** compiler (by lines of code) concatenated into a single file and compiled as a whole program. Note that in this case, all library code (including the Standard Basis Library) was compiled separately. The resulting source file consisted of 32555 lines of code!

The absolute size (in kilobytes) of the generated type and object files is given in figure 9.10.

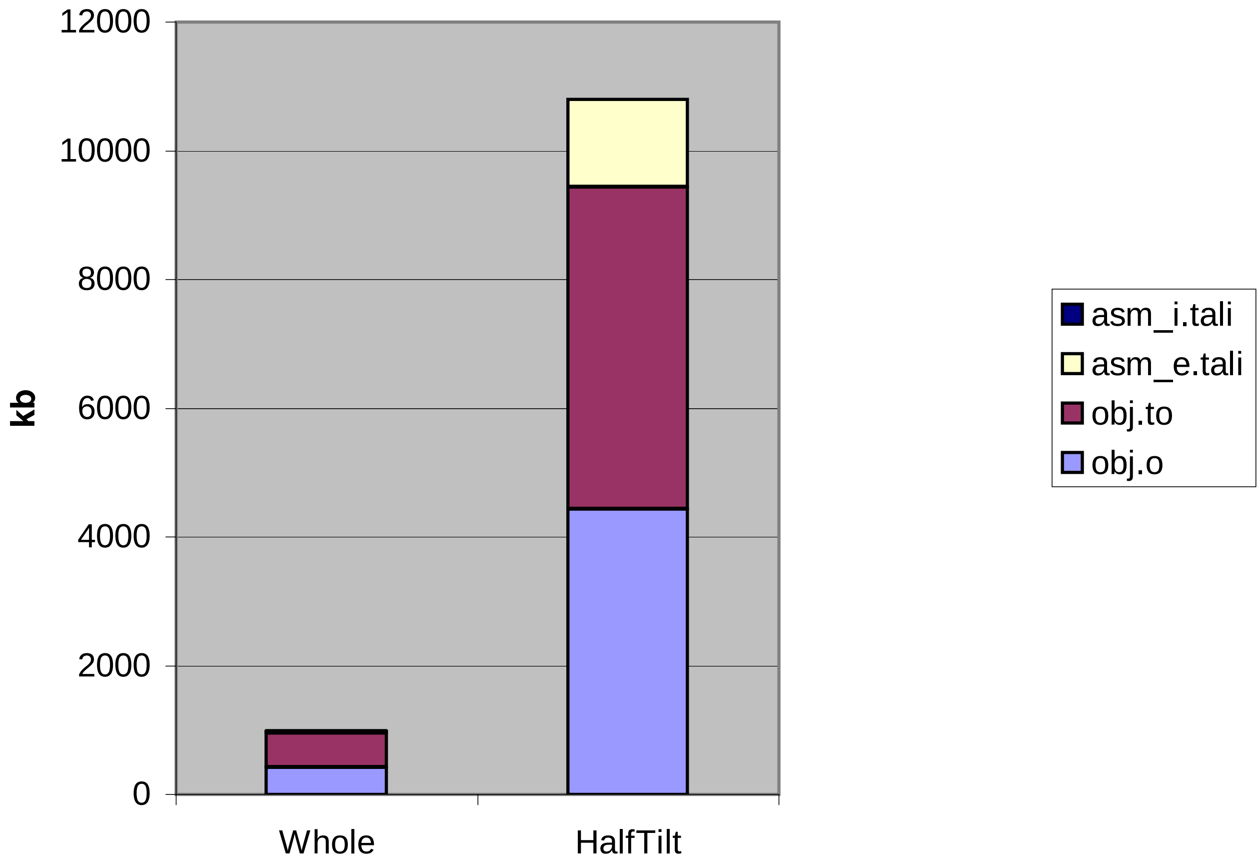



**Figure 9.10:** Absolute sizes of large units (kilobytes).

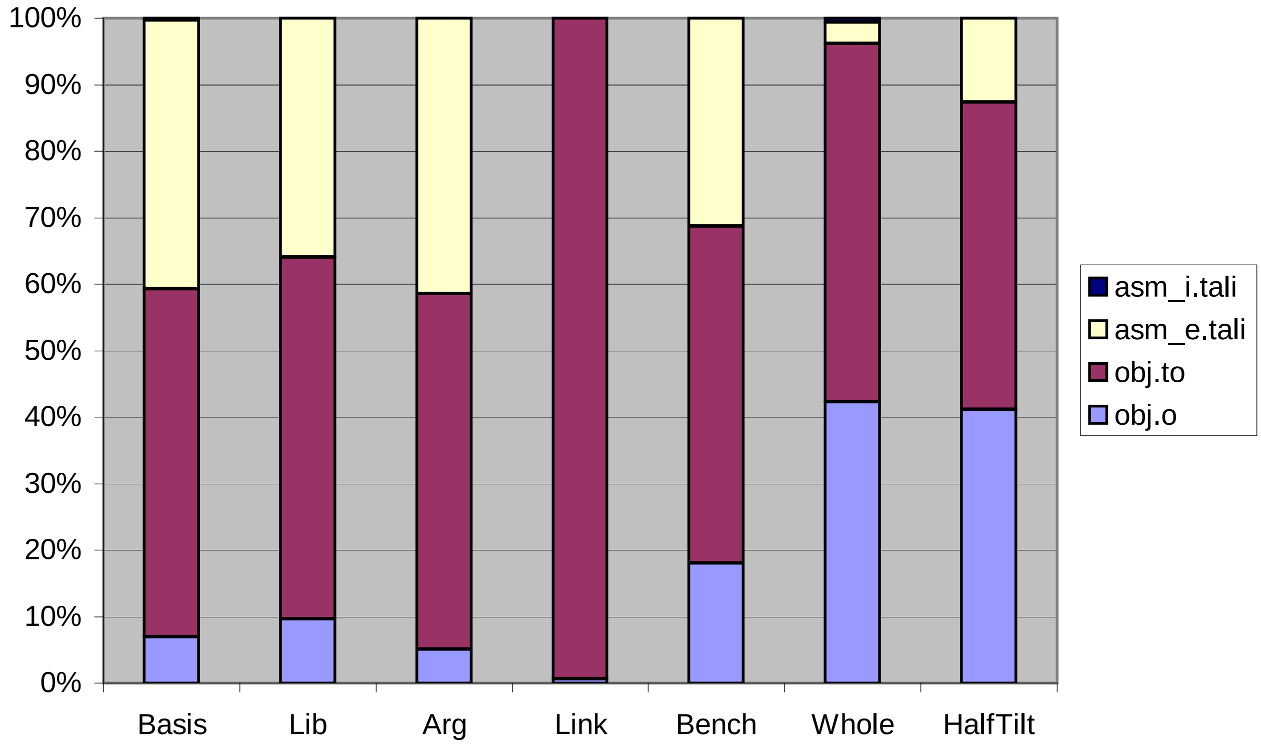



**Figure 9.11:** Relative sizes of large units.

The same data is plotted in figure 9.11 as a percentage of the total, with comparisons to the totals for the separately compiled libraries and benchmarks as shown in figure 9.7. For both of these large programs, the certificate overhead is substantially smaller than the aggregate totals for the other compilation groups, representing approximately 60% of the total size. This is comparable to the overhead for the largest of the benchmark units described in figure 9.5. These two data points combined with figure 9.5 strongly suggest that while there is a noticeable initial overhead for certification, the certificate size remains a relatively constant fraction of the overall size for a large range of program sizes. That is to say, empirically speaking the certificate size seems to grow linearly with the program size.

### 9.8.5 Run time

The **LIL** backend is not designed to be an optimizing backend. Code generation and register allocation are both done in a single linear pass, and no optimizations are done after code generation. Nonetheless, it is useful to measure the runtime performance of the compiled code. It is important that certification not overly restrict the compilation process in such a way as to make efficient code generation impossible. In this section, I argue that even a non-optimizing certifying backend can produce reasonably efficient code.

In order to make this argument, I present comparisons with two other compilers: the Standard ML of New Jersey compiler and the MLton compiler. These comparisons are designed to give some indication of the relative performance of the certifying **TILT** backend with respect to the state of the art in Standard ML compilers. It is important to note however that each of these compilers is fundamentally addressing a very different compilation problem from each of the others. For this reason, this comparison is only really meaningful at the most general high level.

The MLton compiler is designed to produce very efficient code. In order to do this, it compiles only complete programs. The Standard ML of NJ compiler is designed to be used as an interactive system. As such, it supports incremental (but not separate) compilation. It performs significant optimization as well, but is limited somewhat by the needs of incremental compilation and an interactive frontend. Both of these compilers implement precise garbage collection.

The **TILT** compiler on the other hand implements true separate compilation. This greatly limits its ability to optimize code that crosses compilation units. The **TALx86** backend also uses a conservative garbage collector, with a malloc based allocator.

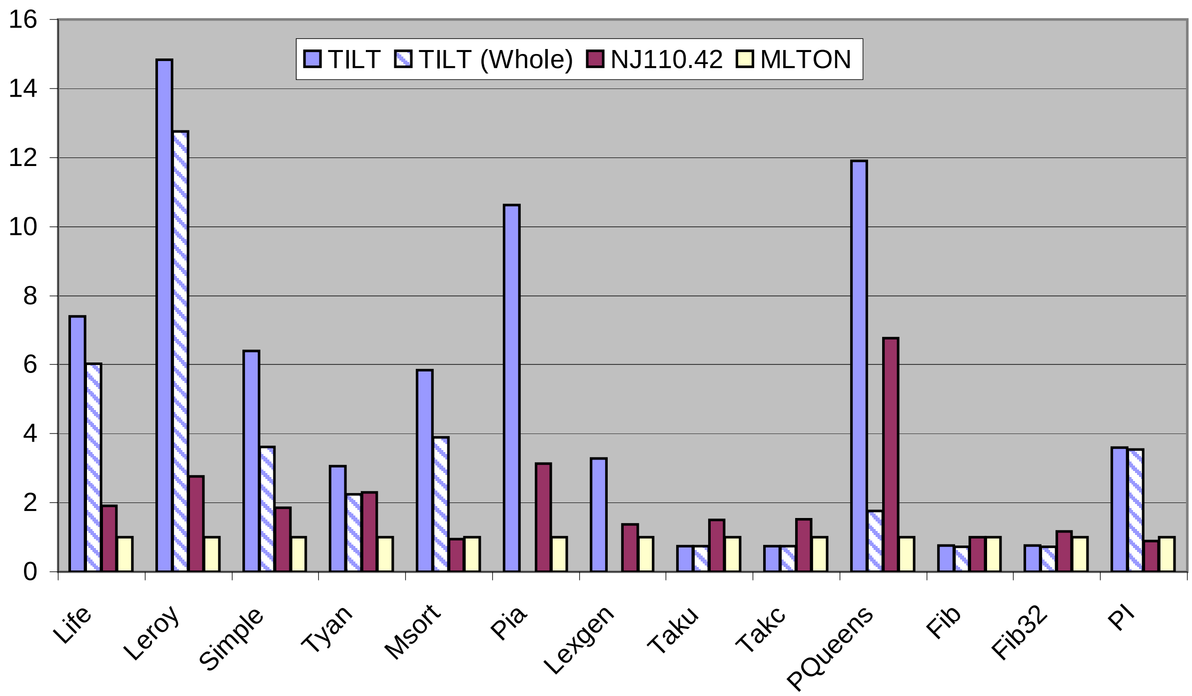



**Figure 9.12:** Normalized runtime (MLton = 1)

Figure 9.12 shows the relative performance of the compiled versions of the **TILT** benchmarks as compiled by each of these three compilers. The results are normalized to the results for the MLton compiler, which is generally the fastest. In order to get some sense of the overhead of seperate compilation, some of the benchmarks were compiled by **TILT** as whole programs as well (e.g. with all of the library code and the Standard Basis Library included into one file). This data is presented in the second column labelled **TILT**(Whole). Because of limitations in the **TILT** foreign function interface however, none of the benchmarks requiring i/o could be compiled as whole programs and so certain of the data points are missing.

Overall, the certifying backend ranges from slightly faster than MLton to almost sixteen times slower, and from approximately twice as fast as SML/NJ to seven times slower. For very small benchmarks, such as the arithmetic benchmarks and the takc/taku benchmarks, **TILT** does quite well. For small benchmarks such as these, there is no penalty for the inability to optimize across compilation units, and the relatively simple register allocation in **TILT** is sufficient. The one exception to this is the pi benchmark, on which **TILT** does quite poorly. This is almost certainly because of the lack of floating point register allocation in **TILT**, since this benchmark is dominated by floating point operations.

The performance of the larger benchmarks is much more varied. In some cases (tyan, lexgen, pqueens) **TILT** is within a factor of two of one or both of the other compilers. In the worst case (leroy) **TILT** is almost sixteen times slower than MLton, and five times slower than SML/NJ. I conjecture that part of this is likely due to separate compilation: all of the larger benchmarks cross compilation unit boundaries a fair bit, via library calls. In addition, several of the benchmarks involve floating point computation, and several of them are fairly allocation intensive. In both of these areas, **TILT** is likely to suffer: from the lack of floating point register allocation in the first case, and from a more expensive memory allocation strategy in the second case.

The whole program version of each **TILT** compiled benchmark is noticeably faster than the separately compiled version: in one particular case (PQueens), faster by a factor of approximately six. This is despite the fact that all of the **TILT** optimizations assume a separate compilation setting even when given whole programs.

# Chapter 10

# Conclusions and future work

## 10.1 Summary

In this dissertation I have shown that certified compilation is possible for full Standard ML, even in the presence of type analysis based optimizations.

### 10.1.1 Theory

To provide a theoretical foundation for this, I defined a series of translations mapping a polymorphically typed lambda calculus extended with type analysis primitives to a typed assembly language. These translations serve to make type analysis explicit in the intermediate representation; to translate uses of functions to uses of closures; and to make control flow and machine state explicit in an abstract assembly language.

I proved each of these translations sound in the sense that each translation maps well-typed terms to well-typed terms. In order to avoid overly constraining the implementation, I also introduced a novel approach for dealing with register allocation, allowing the typing assumptions required for the proof of soundness to be separated from the semantic behavior of the register allocator. In this way, the translation to assembly code remains parametric over the choice of register allocation algorithm, so long as the algorithm chosen satisfies certain minimal typing restrictions.

### 10.1.2 Practice

In order to validate the practicality of my approach, I implemented a certifying backend for the **TILT** compiler using the formal translation as a guide.

The **TILT** middle internal language corresponds closely to the polymorphically typed lambda calculus used as a starting point for the formal compilation process described in this dissertation. For each of the compiled passes described formally in this dissertation, I implemented a corresponding compiler pass in the **TILT** compiler. In addition, I implemented a one pass optimizer to perform simple optimizations to take advantage of the additional program structure exposed by the new translations. The final target of this backend is a slightly modified version of the **TALx86** framework. The final translation to the **TALx86** language follows closely the format of the formal translation in this dissertation, despite the significant syntactic differences between the idealized typed assembly language used in the formal presentation and the **TALx86** language.

Separate compilation of full Standard ML is supported by this new backend, and a large number of programs have been successfully compiled including the entire Standard ML Basis Library and the Standard ML of New Jersey library. In addition I compiled the **TILT** benchmark suite using this typed backend, and measured the results of compilation both in terms of performance as compared to other Standard ML compilers, and in terms of certificate size overhead in the compiled binaries.

### 10.1.3 Conclusions

Certified compilation is possible even for languages as rich as Standard ML, even in the presence of complex type based optimizations. Proving the soundness of typed compiler translations in this setting is feasible. However, there is a trade off between the closeness of the formal model to the implementation and the complexity of the proof process. In this dissertation I attempted to keep a very close correspondence between the formal model and the implementation.

In order to make this more feasible I introduced new techniques for factoring out some inessential choices of the implementation, such as the particular choice of mappings of variables to registers and the method by which this mapping is arrived at. While this particular technique should scale to more complex optimizing code generation, it is likely that maintaining the close correspondence between the formal model and the implementation would become more difficult if more complex code generation and optimization techniques were performed.

The problem of managing certificate size is shown here to be manageable. With no major engineering or tuning of the new **TILT** back end, certificate size overhead was shown to be in general quite reasonable. While the overhead for supporting separate compilation makes up almost half of the aggregate certificate size for the measured programs, simple analysis of the structure of compilation units shows that much of this overhead could be eliminated with a more sophisticated mechanism for sharing type abbreviations across interfaces. Numerous other opportunities exist for eliminating redundant type information, both between units and within units.

In the one case where the certificate overhead was observed to be drastically larger than expected (the link unit discussed in the previous section), this was shown to be due to a failure of the **TALx86** assembler (which otherwise performed admirably) rather than a structural failure of the compiler.

### 10.1.4 Compiler engineering

Maintaining type information on the compiler intermediate forms imposes an additional burden on a compiler writer, just as programming in a typed language imposes an additional burden on a programmer. As is the case with type safe programming languages however, it is becoming increasingly clear that the engineering benefits of typed compilation substantially out-weigh the costs. A type preserving compiler provides a form of automatic self-checking that is extremely valuable to the implementer of the compiler.

Compiler bugs are notoriously difficult to track down and fix, since they often exhibit themselves only as second-order effects. That is, the compiler itself appears to run correctly: it is only in the running of the generated code that incorrect behavior appears. To make matters worse, the problem in the generated code may very well manifest itself not at the incorrect program point, but at some arbitrary later point in the program's run (e.g. because of memory or stack corruption). Matching an incorrect behavior of a generated program to the bug in the compiler that caused it often requires an extensive process of analysis and deduction. By and large, the state of the art in untyped compiler debugging generally relies on carefully stepping through optimized (and hence often obfuscated) code in a debugger.

The great benefit of a type-preserving compiler is that it moves the point of discovery for compiler bugs out of the generated code and into the compiler itself, for a large class of bugs. In other words, many or even most compiler bugs are caught and flagged as soon as they are produced in the compiler, rather than at some arbitrary point in a future run of the generated program. Moreover, if a self-check is run between every compiler pass (at least while in development), the point at which the error is flagged indicates not only the location of the error in the intermediate code but also the particular pass of the compiler that is the most likely culprit (i.e. the pass immediately preceeding the failed self-check).

Of course, type safety does not guarantee correctness. It is still possible for the compiler to generate incorrect code that nonetheless happens to be well-typed. In practice however, this seems to be relatively rare – most errors (and most of the most pernicious errors) tend to be caught. In particular, note that the entire class of errors involving memory corruption are guaranteed to be caught by the type checker! Over the course of developing the certifying backend for **TILT**, my experience was that almost all compiler bugs were caught statically. For example, while developing the register allocator (a notorious source of difficult bugs), all of the register allocation bugs that I encountered were caught by the **TALx86** typechecker.

Most of the bugs that were not caught by the typechecker arose from the more permissive nature of the **TALx86** type system as compared to the Standard ML type system. For example, **TALx86** very reasonably defines 32 bit integer arithmetic to have silent overflow semantics: the compiler is responsible for generating explicit overflow checks and raising exceptions as appropriate. If the compiler fails to emit such as check, the code is still well-typed with respect to the **TALx86** type system: however, its behavior on overflow is incorrect with respect to the semantics of Standard ML. An interesting area for future work in the area of typed assembly languages would be to provide facilities for encoding such source language constraints into the type system (without specializing the type system itself to a particular language).

## 10.2 Future work

This dissertation demonstrates the feasibility of performing certified compilation for Standard ML in a type analysis framework. There remain several directions in which this work could be extended in order to make this more useful and practical.

### 10.2.1 Optimization

The certifying backend implemented as part of this thesis is for the most part not an optimizing one. While some simple optimizations are performed on the **LIL** intermediate code, inspection of the output of the compiler suggests numerous ways in which the intermediate code could be improved (particularly after closure conversion).

Some of these improvements are as simple as extending the **LIL** language with additional constructs. A simple example of such an extension is that of heterogeneous tuples, which would allow the closure converter to avoid boxing and un-boxing 64 bit floating point numbers. Others are more complex: for example, implementing partial redundancy elimination to re-locate closure environment operations used only conditionally.

In addition to improvements in the optimization of the intermediate code, there is considerable room for improvement of the code generator itself. The register allocation technique used is quite simplistic: a more sophisticated algorithm based on graph coloring or graph fusion would produce significantly better code. No floating point register allocation is currently done at all.

The code generator itself is quite limited. No attempt is made at scheduling instructions intelligently, and the instruction selection is fairly ad-hoc. While it does a relatively good job of taking advantage of the CISC nature of the x86 instruction set, a more uniform approach would almost certainly improve the generated code.

The implementation of exceptions in the **LIL** backend is also quite inefficient, both in setting new handlers and in executing handler bodies.

Finally, I believe there is a good deal of work left to be done in tuning and improving the use of type analysis in the **TILT** compiler. No tuning has been done to adjust the parameters to the type analysis optimizations, such as the maximum width record to flatten into registers. Additionally, no work has been done to quantify the actual benefits of type analysis as currently implemented. It would be valuable to measure the effect of these optimizations in isolation, and to use this information to look for additional opportunities to take advantage of the type analysis infrastructure already in place.

### 10.2.2 Garbage collection

The **TALx86** infrastructure assumes the use of a conservative garbage collector. The untyped **TILT** backend makes use of type information at runtime to do precise garbage collection. An interesting topic of future research would be to replace or extend the **TALx86** infrastructure in such a way as to support precise garbage collection using the type information already kept at runtime [VC03].

### 10.2.3 Infrastructure improvements

The **TALx86** certification infrastructure proved impressively flexible in serving as a target for the **TILT** compiler with a minimum of modification. However, the performance of the **TALx86** type checker could be improved upon substantially. Currently, assembling and typechecking large units takes dramatically longer than the entire process of compilation within **TILT** (including numerous internal type checks). While the type system is more detailed and low level at the **TALx86** level, many of the techniques discussed in chapter 9 used to improve the performance of the **LIL** type checker are still applicable.

In addition, as discussed in section 9.8.4 there are a few important cases where the **TALx86** assembler seems to fail to preserve the physical sharing present in the original typed assembly source file which degrade its performance on certain units immensely.

### 10.2.4 Reducing the trusted computing base

Another concern with the **TALx86** infrastructure is that there is no proof of type safety for the language as implemented. This is of significant concern in an actual certifying compilation system, since in the absence of such a proof, even the correctness of the **TALx86** type checker does not necessarily imply the safety of the programs it certifies. And of course, there is no guarantee that the implemented type checker faithfully and soundly implements the static semantics of the type system for the language.

Attempts have recently been made to address both of these problems by providing certified code infrastructures that both implement a language with a formal proof of type safety, and which try to reduce the complexity of the checking problem as much as possible so as to minimize the amount of trusted code in the certifier [Cra03, App01]. A valuable area of future research would be to re-target the certifying **TILT** backend to such a system.

## 10.3 Conclusions

This dissertation has clearly shown that certifying compilation is feasible for a rich language like Standard ML, even in the presence of type analysis and other other advanced optimizations. The problem of controlling the intermediate size of programs and of controlling the certificate overhead on the generated code is almost certainly tractable. Even in the system implemented in this thesis, which made no effort to optimize for these properties beyond the minimum necessary to demonstrate feasibility, the results were easily within reach of being sufficient for a practical and usable system.

The problem of connecting formal models of compiler translations to their actual implementations remains a difficult one. As the formal models scale up to more closely model the actual languages and transformations implemented in the compiler, the syntactic overhead and complexity of proofs about the model increases significantly to the point that confidence in the correctness of the proofs must inevitably fall. New techniques for dealing with this complexity are needed, and while this dissertation attempts to address this to a certain extent, in general it remains an open problem. One promising approach to dealing with this lies in the use of logical frameworks to mechanically check proofs, or even to assist in generating them.

Certifying compilation is the natural extension of typed compilation. In this role, a certifying compiler greatly increases the ability of the compiler writer to write a correct compiler by extending the benefits of type checking to a lower level. As with all disciplines, the discipline of having always to consider the type correctness of compiler transformations is sometimes burdensome. I believe that it is also a valuable one. It should always be clear to the compiler writer why a particular transformation is safe. The type checker in a type preserving compiler enforces this discipline. It is important that the flexibility of type systems for compiler internal languages continue to be improved upon, so that the cases where the type checker must reject safe code due to its own limitations can be made increasingly rare.

Most importantly, certifying compilation provides an important tool for coming to grips with the increasing problem of providing security in a wide-open, networked universe. Delivering security along with downloaded code is an essential part of providing value to the end user. This thesis demonstrates that automatically generating such security in the form of certified code is well within our grasp.

# Appendix A

# MIL static semantics

**Well-formed Kind** $\boxed{\vdash \kappa\ \mathbf{ok}}$

$$\frac{}{\vdash \mathrm{T}_{32}\ \mathbf{ok}} \qquad \frac{\vdash \kappa_1\ \mathbf{ok} \quad \vdash \kappa_2\ \mathbf{ok}}{\vdash \kappa_1 \to \kappa_2\ \mathbf{ok}} \qquad \frac{\vdash \kappa_1\ \mathbf{ok} \quad \vdash \kappa_2\ \mathbf{ok}}{\vdash \kappa_1 \times \kappa_2\ \mathbf{ok}}$$

**Well-formed type context** $\boxed{\vdash \Delta\ \mathbf{ok}}$

$$\frac{}{\vdash \bullet\ \mathbf{ok}} \qquad \frac{\vdash \Delta\ \mathbf{ok} \quad \Delta \vdash \kappa\ \mathbf{ok}}{\vdash \Delta, \alpha{:}\kappa\ \mathbf{ok}}\ \alpha \notin \Delta$$

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

$$\frac{\vdash \Delta, \alpha{:}\kappa, \Delta'\ \mathbf{ok}}{\Delta, \alpha{:}\kappa, \Delta' \vdash \alpha : \kappa} \qquad \frac{\vdash \Delta\ \mathbf{ok}}{\Delta \vdash \mathtt{Int} : \mathrm{T}_{32}} \qquad \frac{\vdash \Delta\ \mathbf{ok}}{\Delta \vdash \mathtt{Boxf} : \mathrm{T}_{32}} \qquad \frac{\vdash \Delta\ \mathbf{ok}}{\Delta \vdash \mathtt{Farray} : \mathrm{T}_{32}}$$

$$\frac{\vdash \Delta\ \mathbf{ok}}{\Delta \vdash \mathtt{Exn} : \mathrm{T}_{32}} \qquad \frac{\Delta \vdash c : \mathrm{T}_{32}}{\Delta \vdash \mathtt{Array}_c : \mathrm{T}_{32}} \qquad \frac{\Delta \vdash c : \mathrm{T}_{32}}{\Delta \vdash \mathtt{Dyntag}_c : \mathrm{T}_{32}}$$

$$\frac{\Delta \vdash c_1 : \mathrm{T}_{32} \quad \Delta \vdash c_2 : \mathrm{T}_{32}}{\Delta \vdash \mathtt{Vararg}_{c_1 \to c_2} : \mathrm{T}_{32}} \qquad \frac{\Delta, \alpha{:}\mathrm{T}_{32}, \beta{:}\mathrm{T}_{32} \vdash c_1 : \mathrm{T}_{32} \quad \Delta, \alpha{:}\mathrm{T}_{32}, \beta{:}\mathrm{T}_{32} \vdash c_2 : \mathrm{T}_{32}}{\Delta \vdash \mu(\alpha, \beta).(c_1, c_2) : \mathrm{T}_{32} \times \mathrm{T}_{32}}\ \alpha, \beta \notin \Delta$$

$$\frac{\begin{array}{c}\Delta \vdash c_i : \mathrm{T}_{32} \quad i \in 0 \ldots j \\ \Delta \vdash c : \mathrm{T}_{32}\end{array}}{\Delta \vdash (c_0 \ldots c_j) \to c : \mathrm{T}_{32}} \qquad \frac{\Delta \vdash c_i : \mathrm{T}_{32} \quad i \in 0 \ldots j}{\Delta \vdash c_1 \times \ldots \times c_j : \mathrm{T}_{32}}$$

$$\frac{\Delta \vdash c_i : \mathrm{T}_{32} \quad i \in 0 \ldots j \qquad \Delta \vdash c : \mathrm{T}_{32}}{\Delta \vdash \texttt{Sum}_n(c_0 \ldots c_j) : \mathrm{T}_{32}} \qquad \frac{\Delta \vdash c_i : \mathrm{T}_{32} \quad i \in 0 \ldots j \qquad \Delta \vdash c : \mathrm{T}_{32} \quad m \leq n + j}{\Delta \vdash \texttt{Sum}_n^m(c_0 \ldots c_j) : \mathrm{T}_{32}}$$

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

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

**Base constructors** $\square$

The base constructors consist of `Int`, `Boxf`, arrows, records, projections from mu types, `Vararg`, `Sum`, `Array`, `Farray`, `Exn`, and `Dyntag`. The non-record base constructors consist of any of the above except record types.

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

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

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

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

$$\frac{\vdash \kappa_1 \ \mathbf{ok} \qquad \Delta, \alpha{:}\kappa_1 \vdash c_1 : \kappa_2 \quad \Delta \vdash c_2 : \kappa_1}{\Delta \vdash (\lambda(\alpha{:}\kappa_1).c_1)c_2 \equiv c_1[c_2/\alpha] : \kappa_2}$$

$$\frac{\vdash \kappa_1 \ \mathbf{ok} \qquad \Delta, \alpha{:}\kappa_1 \vdash c_1 \equiv c_2 : \kappa_2}{\Delta \vdash \lambda(\alpha{:}\kappa_1).c_1 \equiv \lambda(\alpha{:}\kappa_1).c_2 : \kappa_1 \to \kappa_2} \qquad \frac{\Delta \vdash c_1 \equiv c_1' : \kappa_1 \to \kappa_2 \qquad \Delta \vdash c_2 \equiv c_2' : \kappa_1}{\Delta \vdash c_1 c_2 \equiv c_1' c_2' : \kappa_2}$$

$$\frac{\Delta \vdash c_1\ \mathbf{ok} \quad \Delta \vdash c_2\ \mathbf{ok} \quad \Delta \vdash c_1 \equiv \times[c'_1, \ldots, c'_n] : \mathrm{T}_{32} \quad n \leq \text{flattenlimit}}{\Delta \vdash \mathtt{Vararg}_{c_1 \to c_2} \equiv (c'_1, \ldots, c'_n) \to c_2 : \mathrm{T}_{32}}$$

$$\frac{\Delta \vdash c_1\ \mathbf{ok} \quad \Delta \vdash c_2\ \mathbf{ok} \quad \Delta \vdash c_1 \equiv \times[c'_1, \ldots, c'_n] : \mathrm{T}_{32} \quad n > \text{flattenlimit}}{\Delta \vdash \mathtt{Vararg}_{c_1 \to c_2} \equiv c_1 \to c_2 : \mathrm{T}_{32}}$$

$$\frac{\begin{array}{c}\Delta \vdash c_1\ \mathbf{ok} \quad \Delta \vdash c_2\ \mathbf{ok} \quad \Delta \vdash c_1 \equiv c'_1 : \mathrm{T}_{32} \\ \text{where } c'_1 \text{ is a non-record base constructor}\end{array}}{\Delta \vdash \mathtt{Vararg}_{c_1 \to c_2} \equiv c_1 \to c_2 : \mathrm{T}_{32}}$$

$$\frac{\Delta \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_1\, c \equiv \pi_1\, c' : \kappa_1} \qquad \frac{\Delta \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_2\, c \equiv \pi_2\, c' : \kappa_2}$$

**Type equivalence** $\boxed{\Delta \vdash \tau \equiv \tau' : \mathrm{T}_{32}}$

$$\frac{\Delta \vdash c \equiv c' : \mathrm{T}_{32}}{\Delta \vdash T(c) \equiv T(c') : \mathrm{T}_{32}} \qquad \frac{\Delta \vdash \tau_i \equiv \tau'_i : \mathrm{T}_{32} \quad i \in 1 \ldots n}{\Delta \vdash \tau_1 \times \ldots \times \tau_n \equiv \tau'_1 \times \ldots \times \tau'_n : \mathrm{T}_{32}}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau_i \equiv \tau'_i : \mathrm{T}_{32} \quad i \in 1 \ldots n \\ \Delta \vdash \tau \equiv \tau' : \mathrm{T}_{32}\end{array}}{\Delta \vdash \forall[\alpha{::}\kappa_1, \ldots, \alpha{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau \equiv \forall[\alpha{::}\kappa_1, \ldots, \alpha{::}\kappa_n](\tau'_1, \ldots, \tau'_m)(k) \to \tau' : \mathrm{T}_{32}}$$

**Well-formed float value** $\boxed{\Delta; \Gamma \vdash fv : \mathtt{Float}}$

$$\frac{\vdash \Delta\ \mathbf{ok} \quad \Delta \vdash \Gamma\ \mathbf{ok}}{\Delta; \Gamma \vdash r : \mathtt{Float}}\ r$$

$$\frac{\vdash \Delta\ \mathbf{ok} \quad \Delta \vdash \Gamma[x_f]\ \mathbf{ok}}{\Delta; \Gamma[x_f] \vdash x_f : \mathtt{Float}}\ fvar$$

**Well-formed small value** $\boxed{\Delta;\Gamma \vdash sv : \tau}$

$$\frac{\Delta \vdash \Gamma[x{:}\tau]\ \mathbf{ok}}{\Delta;\Gamma[x{:}\tau] \vdash x : \tau}\ \mathit{var}$$

$$\frac{\Delta \vdash \Gamma\ \mathbf{ok}}{\Delta;\Gamma \vdash i : \mathtt{Int}}\ \mathit{int}$$

$$\frac{\Delta;\Gamma \vdash sv : c_i[\pi_1\,\mu(\alpha,\beta)(c_1,c_2)/\alpha, \pi_2\,\mu(\alpha,\beta)(c_1,c_2)/\beta]}{\Delta;\Gamma \vdash \mathtt{roll}_{\pi_i\,\mu(\alpha,\beta)(c_1,c_2)}\ sv : \pi_i\,\mu(\alpha,\beta)(c_1,c_2)}\ \mathit{roll}$$

$$\frac{\Delta \vdash c \equiv \pi_i\,\mu(\alpha,\beta)(c_1,c_2) : \mathrm{T}_{32} \quad \Delta;\Gamma \vdash sv : \pi_i\,\mu(\alpha,\beta)(c_1,c_2)}{\Delta;\Gamma \vdash \mathtt{unroll}_c\ sv : c_i[\pi_1\,\mu(\alpha,\beta)(c_1,c_2)/\alpha, \pi_2\,\mu(\alpha,\beta)(c_1,c_2)/\beta]}\ \mathit{unroll}$$

$$\frac{\Delta \vdash \mathtt{Sum}_i^j(\vec{c}) : \mathrm{T}_{32}}{\Delta;\Gamma \vdash \mathtt{inj\_tag}^j_{\mathtt{Sum}_i(\vec{c})} : \mathtt{Sum}_i(\vec{c})}\ \mathit{inj\_tag}$$

**Well formed 32 bit instructions** $\boxed{\Delta;\Gamma \vdash opr : \tau}$

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

$$\frac{\Delta;\Gamma \vdash fv : \mathtt{Float}}{\Delta;\Gamma \vdash \mathtt{boxf}\ fv : \mathtt{Boxf}}\ \mathit{boxf}$$

$$\frac{\Delta;\Gamma \vdash sv_i : \tau_i}{\Delta;\Gamma \vdash \langle sv_1, \ldots, sv_n\rangle : \tau_1 \times \ldots \times \tau_n}\ \mathit{tuple}$$

$$\frac{\Delta;\Gamma \vdash sv : c_1 \to c_2}{\Delta;\Gamma \vdash \mathtt{vararg}_{c_1 \to c_2}\ sv : \mathtt{Vararg}_{c_1 \to c_2}}\ \mathit{vararg}$$

$$\frac{\Delta;\Gamma \vdash sv : \mathtt{Vararg}_{c_1 \to c_2}}{\Delta;\Gamma \vdash \mathtt{onearg}_{c_1 \to c_2}\, sv : c_1 \to c_2}\ \mathit{onearg}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv : \forall[\alpha_1{::}\kappa_1, \ldots, \alpha_n{::}\kappa_n](\tau_1, \ldots, \tau_m)(k) \to \tau \\ \Delta;\Gamma \vdash sv_i : \tau_i[c_1/\alpha_1, \ldots, c_n/\alpha_n] \\ \Delta;\Gamma \vdash fv_i : \mathtt{Float}\end{array}}{\Delta;\Gamma \vdash sv[c_1, \ldots, c_n](sv_1, \ldots, sv_m)(fv_1, \ldots, fv_k) : \tau[c_1/\alpha_1, \ldots, c_n/\alpha_n]}\ \mathit{app}$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{Sum}_i^j(\vec{c}) : \mathrm{T}_{32} \\ \Delta;\Gamma \vdash sv : \mathtt{Sum}_i^j(\vec{c})\end{array}}{\Delta;\Gamma \vdash \mathtt{proj}^j_{\mathtt{Sum}_i(\vec{c})}\, sv : c_j}\ \mathit{proj}$$

$$\frac{\Delta \vdash \mathtt{Sum}_i^j(\vec{c}) : \mathrm{T}_{32} \quad \Delta;\Gamma \vdash sv : c_j}{\Delta;\Gamma \vdash \mathtt{inj}^j_{\mathtt{Sum}_i(\vec{c})}\, sv : \mathtt{Sum}_i(\vec{c})}\ \mathit{inj}$$

$$\frac{\Delta;\Gamma \vdash sv : \tau_1 \times \ldots \times \tau_n}{\Delta;\Gamma \vdash \mathtt{select}^i\, sv : \tau_i}\ \mathit{select}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau : \mathrm{T}_{32} \\ \Delta;\Gamma \vdash sv : \mathtt{Sum}_i(\vec{c}) \quad \Delta;\Gamma[x_j : \mathtt{Sum}_i^j(\vec{c})] \vdash e_j : \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{case}_\tau(sv)\,(x_1.e_1, \ldots, x_n.e_n) : \tau}\ \mathit{case}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash sv : \mathtt{Exn} \Delta \vdash \tau : \mathrm{T}_{32} \\ \Delta;\Gamma \vdash sv_1 : \mathtt{Dyntag}_{c_1} \quad \Delta;\Gamma[x_1{:}c_1] \vdash e_1 : \tau \quad \Delta;\Gamma \vdash e_2 : \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{exncase}_\tau(sv)\,(sv_1 \Rightarrow x_1.e_1, \_ \Rightarrow e_2) : \tau}\ \mathit{exncase}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \quad \Delta;\Gamma[x : Exn] \vdash e_2 : \tau}{\Delta;\Gamma \vdash \mathtt{handle}_\tau(e_1, x.e_2) : \tau}\ \mathit{handle}$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \mathtt{Dyntag}_c \quad \Delta;\Gamma \vdash sv_2 : c}{\Delta;\Gamma \vdash \mathtt{inj\_dyn}_c(sv_1, sv_2) : \mathtt{Exn}}\ \mathit{inj\_exn}$$

$$\frac{\Delta \vdash c : \mathrm{T}_{32} \quad \Delta;\Gamma \vdash sv_1 : \mathtt{Array}_c \quad \Delta;\Gamma \vdash sv_2 : \mathtt{Int}}{\Delta;\Gamma \vdash \mathtt{sub}_c(sv_1, sv_2) : c}\ sub$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \mathtt{Int} \quad \Delta;\Gamma \vdash sv_2 : c}{\Delta;\Gamma \vdash \mathtt{array}_c(sv_1, sv_2) : \mathtt{Array}_c}\ array$$

$$\frac{\Delta;\Gamma \vdash sv : \mathtt{Int} \quad \Delta \vdash fv : \mathtt{Float}}{\Delta;\Gamma \vdash \mathtt{farray}(sv, fv) : \mathtt{Farray}}\ farray$$

$$\frac{\Delta \vdash \tau : \mathrm{T}_{32} \quad \Delta;\Gamma \vdash sv : \mathtt{Exn}}{\Delta;\Gamma \vdash \mathtt{raise}_\tau\ sv : \tau}\ raise$$

$$\frac{\Delta \vdash c : \mathrm{T}_{32} \quad \Delta \vdash \Gamma\ \mathbf{ok}}{\Delta;\Gamma \vdash \mathtt{mkexntag}_c : \mathtt{Dyntag}_c}\ mkexntag$$

**Well formed float instructions** $\boxed{\Delta;\Gamma \vdash opr : \mathtt{Float}}$

$$\frac{\Delta;\Gamma \vdash sv : \mathtt{Boxf}}{\Delta;\Gamma \vdash \mathtt{unboxf}\ sv : \mathtt{Float}}\ unbox$$

$$\frac{\Delta;\Gamma \vdash sv_1 : \mathtt{Farray} \quad \Delta;\Gamma \vdash sv_2 : \mathtt{Int}}{\Delta;\Gamma \vdash \mathtt{fsub}(sv_1, sv_2) : \mathtt{Float}}\ fsub$$

$$\frac{\Delta;\Gamma \vdash fv : \mathtt{Float}}{\Delta;\Gamma \vdash fv : \mathtt{Float}}\ fv$$

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

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

$$\frac{\Delta;\Gamma \vdash i : \tau' \quad \Delta;\Gamma[x : \tau'] \vdash e : \tau}{\Delta;\Gamma \vdash \mathtt{let}_\tau\, x = i\, \mathtt{in}\, e : \tau}\ i$$

$$\frac{\Delta;\Gamma \vdash i : \mathtt{Float} \quad \Delta;\Gamma[x_f] \vdash e : \tau}{\Delta;\Gamma \vdash \mathtt{let}_\tau\, x_f = i\, \mathtt{in}\, e : \tau}\ i64$$

$$\frac{\begin{array}{c}\Delta;\Gamma[f : \forall[\vec{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r, \vec{\alpha::\kappa}, \vec{x:\tau}, \vec{x_f}] \vdash e_f : \tau_r \\ \Delta;\Gamma[f : \forall[\vec{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau_r] \vdash e : \tau\end{array}}{\Delta;\Gamma \vdash \mathtt{let}_\tau\, \mathtt{rec}_{\tau_r}\, f[\vec{\alpha::\kappa}](\vec{x:\tau})(\vec{x_f}).e_f\, \mathtt{in}\, e : \forall[\vec{\alpha::\kappa}](\vec{\tau})(|x_f|) \to \tau}\ rec$$

**Well-formed Context** $\boxed{\Delta \vdash \Gamma\ \mathbf{ok}}$

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

# Appendix B

# LIL static semantics

**Definitions** $\square$

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

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

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

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

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

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

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

**Well-formed type and kind ontext** $\boxed{\vdash \Delta\ \mathbf{ok}}$

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

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

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

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

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

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

$$\frac{\Delta \vdash c : \kappa_i \quad (i \in \{1 \ldots n\})}{\Delta \vdash \texttt{inj}_i^{+[\kappa_1,\ldots,\kappa_n]}\, c : +[\kappa_1, \ldots, \kappa_n]} \qquad \frac{\begin{array}{c}\Delta \vdash c : +[\kappa_1, \ldots, \kappa_n] \\ \Delta, \alpha_i{:}\kappa_i \vdash c_i : \kappa \quad i \in 1 \ldots n\end{array}}{\Delta \vdash \texttt{case}\, c[(\alpha_1.c_1, \ldots, \alpha_n.c_n)] : \kappa}\ \alpha_i \notin \Delta$$

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

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

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

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

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

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

$$\frac{\Delta \vdash c : +[\kappa_1, \ldots, \kappa_n]}{\Delta \vdash \mathtt{case}(c, [\alpha_1.\, \mathtt{inj}_1^{+[\kappa_1,\ldots,\kappa_n]}\, \alpha_1, \ldots, \alpha_n.\, \mathtt{inj}_n^{+[\kappa_1,\ldots,\kappa_n]}\, \alpha_n]) \equiv c : +[\kappa_1, \ldots, \kappa_n]}$$

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

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

$$\frac{\begin{array}{c}\Delta \vdash c : \kappa_i \quad \Delta \vdash \kappa_j\ \mathbf{ok} \quad j \in 1 \ldots n \\ \Delta, \alpha_j{:}\kappa_j \vdash c_j : \kappa \quad j \in 1 \ldots n\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_i^{+[\kappa_1,\ldots,\kappa_n]}\, c, [\ldots, \alpha_i.c_i, \ldots]) \equiv c_i[c/\alpha_i] : \kappa}$$

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

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

$$\frac{\Delta \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_1\, c \equiv \pi_1\, c' : \kappa_1} \qquad \frac{\Delta \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Delta \vdash \pi_2\, c \equiv \pi_2\, c' : \kappa_2}$$

$$\frac{\Delta \vdash c \equiv c' : \kappa_i \quad \Delta \vdash \kappa_j \ \mathbf{ok} \quad j \in 1 \ldots n}{\Delta \vdash \mathtt{inj}_i^{+[\kappa_1,\ldots,\kappa_n]}\, c \equiv \mathtt{inj}_i^{+[\kappa_1,\ldots,\kappa_n]}\, c' : +[\kappa_1, \ldots, \kappa_n]}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv c' : +[\kappa_1, \ldots, \kappa_n] \\ \Delta, \alpha_i{:}\kappa_i \vdash c_i \equiv c'_i : \kappa \quad i \in 1 \ldots n\end{array}}{\Delta \vdash \mathtt{case}(c, [\alpha_1.c_1, \ldots, \alpha_n.c_n]) \equiv \mathtt{case}(c', [\alpha_1.c'_1, \ldots, \alpha_n.c'_n]) : \kappa}$$

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

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

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

**Well-formed term context** $\boxed{\Delta \vdash \Gamma \ \mathbf{ok}}$

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

**Well-formed heap context** $\boxed{\vdash \Psi \ \mathbf{ok}}$

$$\frac{}{\vdash \bullet \ \mathbf{ok}} \qquad \frac{\vdash \Psi \ \mathbf{ok} \quad \bullet \vdash \tau : \mathrm{T}_{32}}{\vdash \Psi, \ell{:}\tau \ \mathbf{ok}}\ \ell \notin \Gamma$$

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

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

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

$$\frac{\Delta \vdash \Gamma, x{:}\tau, \Gamma' \ \mathbf{ok} \quad \vdash \Psi \ \mathbf{ok}}{\Psi;\Delta;\Gamma, x{:}\tau, \Gamma' \vdash x : \tau} \qquad \frac{\Delta \vdash \Gamma \ \mathbf{ok} \quad \vdash \Psi \ \mathbf{ok}}{\Psi;\Delta;\Gamma \vdash i : \mathtt{Int}} \qquad \frac{\Delta \vdash \Gamma \ \mathbf{ok} \quad \vdash \Psi_1, \ell{:}\tau, \Psi_2 \ \mathbf{ok}}{\Psi_1, \ell{:}\tau, \Psi_2;\Delta;\Gamma \vdash \ell : \tau}$$

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

$$\frac{\Psi;\Delta;\Gamma \vdash sv_1 : \mathtt{Dyntag}\,\tau \quad \Psi;\Delta;\Gamma \vdash sv_2 : \tau}{\Psi;\Delta;\Gamma \vdash \mathtt{inj\_dyn}_\tau(sv_1, sv_2) : \mathtt{Dyn}} \qquad \frac{\Delta \vdash c \equiv \bigvee[\ldots, c_i, \ldots] : \mathrm{T}_{32} \quad \Psi;\Delta;\Gamma \vdash sv : c_i}{\Psi;\Delta;\Gamma \vdash \mathtt{inj\_union}_c\, sv : c}$$

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

$$\frac{\Delta \vdash \Gamma \ \mathbf{ok} \quad \vdash \Psi \ \mathbf{ok}}{\Psi;\Delta;\Gamma \vdash \mathtt{tag}_i : \mathtt{Tag}(\bar{i})}$$

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

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

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

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

$$\frac{\Psi;\Delta;\Gamma \vdash sv : \tau}{\Psi;\Delta;\Gamma \vdash sv : \tau \ \mathbf{opr}_{32}} \qquad \frac{\Psi;\Delta;\Gamma \vdash sv : \times(\tau_0{::}\ldots{::}\tau_i{::}c')}{\Psi;\Delta;\Gamma \vdash \mathtt{select}^i\, sv : \tau_i \ \mathbf{opr}_{32}}$$

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

$$\frac{\Psi;\Delta;\Gamma\vdash sv:\bigvee[\tau_0,\ldots,\tau_k]\quad \Psi;\Delta;\Gamma,x_i{:}\tau_i\vdash e_i:\tau\ \mathbf{exp} \\ \Delta\vdash\tau_i\equiv\mathtt{Tag}(i):\mathrm{T}_{32}\quad i\in 0\ldots(j-1) \\ \Delta\vdash\tau_i\equiv\times[\mathtt{Tag}(i),\tau_i']:\mathrm{T}_{32}\quad i\in j\ldots k}{\Psi;\Delta;\Gamma\vdash\mathtt{case}(sv)(x.e_0,\ldots,x.e_k):\tau\ \mathbf{opr}_{32}}$$

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

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

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

$$\frac{\Psi;\Delta;\Gamma\vdash sv:\ \rightarrow([\tau_0,\ldots,\tau_n])([\phi_0,\ldots,\phi_k])(\tau) \\ \Psi;\Delta;\Gamma\vdash sv_i:\tau_i\quad \Psi;\Delta;\Gamma\vdash fv_i:\phi_i}{\Psi;\Delta;\Gamma\vdash sv(sv_0,\ldots,sv_n)(fv_0,\ldots,fv_k):\tau\ \mathbf{opr}_{32}}$$

$$\frac{\Psi;\Delta;\Gamma\vdash sv:\mathtt{Code}[\tau_0,\ldots,\tau_n][\phi_0,\ldots,\phi_k](\tau) \\ \Psi;\Delta;\Gamma\vdash sv_i:\tau_i\quad \Psi;\Delta;\Gamma\vdash fv_i:\phi_i}{\Psi;\Delta;\Gamma\vdash\mathtt{call}\,sv(sv_0,\ldots,sv_n)(fv_0,\ldots,fv_k):\tau\ \mathbf{opr}_{32}}$$

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma\vdash opr : \tau\ \textbf{opr}_{32}\\ \Psi;\Delta;\Gamma, x{:}\tau\vdash e : \tau'\ \textbf{exp}\end{array}}{\Psi;\Delta;\Gamma\vdash \texttt{let}\ x = opr\ \texttt{in}\ e : \tau'\ \textbf{exp}} \qquad \frac{\begin{array}{c}\Psi;\Delta;\Gamma\vdash opr : \phi\ \textbf{opr}_{32}\\ \Psi;\Delta;\Gamma, x_{64}{:}\phi\vdash e : \tau'\ \textbf{exp}\end{array}}{\Psi;\Delta;\Gamma\vdash \texttt{let}\ x_{64} = fopr\ \texttt{in}\ e : \tau'\ \textbf{exp}}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma\vdash sv : \exists[\kappa](c)\\ \Psi;\Delta,\alpha{:}\kappa;\Gamma, x{:}(c\alpha)\vdash e : \tau'\ \textbf{exp} \quad \alpha\notin fv(\tau')\end{array}}{\Psi;\Delta;\Gamma\vdash \texttt{let}[\alpha, x] = \texttt{unpack}\ sv\ \texttt{in}\ e : \tau'\ \textbf{exp}}\ \alpha, x\notin\Delta,\Gamma$$

$$\frac{\begin{array}{c}\Psi;\Delta,\beta{:}\kappa_1,\gamma{:}\kappa_2,\Delta';\Gamma[\langle\beta,\gamma\rangle/\alpha]\vdash e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha]\ \textbf{exp}\\ \Delta,\alpha{:}\kappa_1\times\kappa_2,\Delta'\vdash c\equiv\alpha : \kappa_1\times\kappa_2\end{array}}{\Psi;\Delta,\alpha{:}\kappa_1\times\kappa_2,\Delta';\Gamma\vdash \texttt{let}\langle\beta,\gamma\rangle = c\ \texttt{in}\ e : \tau\ \textbf{exp}}\ \beta,\gamma\notin\Delta$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma\vdash e[c_1, c_2/\beta,\gamma] : \tau\ \textbf{exp}\\ \Delta\vdash c\equiv\langle c_1, c_2\rangle : \kappa_1\times\kappa_2\end{array}}{\Psi;\Delta;\Gamma\vdash \texttt{let}\langle\beta,\gamma\rangle = c\ \texttt{in}\ e : \tau\ \textbf{exp}}\ \beta\notin\Delta$$

$$\frac{\begin{array}{c}\Psi;\Delta,\beta{:}\kappa[\mu j.\kappa/j],\Delta';\Gamma[\texttt{fold}_{\mu j.\kappa}\,\beta/\alpha]\vdash e[\texttt{fold}_{\mu j.\kappa}\,\beta/\alpha] : \tau[\texttt{fold}_{\mu j.\kappa}\,\beta/\alpha]\ \textbf{exp}\\ \Delta,\alpha{:}\mu j.\kappa,\Delta'\vdash c\equiv\alpha : \mu j.\kappa\end{array}}{\Psi;\Delta,\alpha{:}\mu j.\kappa,\Delta';\Gamma\vdash \texttt{let}(\texttt{fold}\,\beta) = c\ \texttt{in}\ e : \tau\ \textbf{exp}}$$

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

$$\frac{\begin{array}{c}\Delta, \alpha{:}\kappa_1 + \kappa_2, \Delta' \vdash c \equiv \alpha : \kappa_1 + \kappa_2 \quad \alpha_1, \alpha_2 \notin \Delta, \Delta' \\ \Psi; \Delta, \beta{:}\kappa_i, \Delta'; \Gamma[\mathtt{inj}_i\,\beta/\alpha] \vdash e[\mathtt{inj}_i\,\beta/\alpha] : \tau[\mathtt{inj}_i\,\beta/\alpha] \ \mathbf{exp} \\ \Psi; \Delta, \beta{:}\kappa_j, \Delta'; \Gamma[\mathtt{inj}_j\,\beta/\alpha] \vdash sv[\mathtt{inj}_j\,\beta/\alpha] : \mathtt{Void} \quad j \in 1 \ldots i-1, i+1 \ldots n\end{array}}{\Psi; \Delta, \alpha{:} +[\kappa_1, \ldots, \kappa_n], \Delta' \vdash \mathtt{let}_\tau\,\mathtt{inj}_i\,\beta = (c, sv)\,\mathtt{in}\,e : \tau \ \mathbf{exp}}$$

$$\frac{\begin{array}{c}\Delta \vdash c \equiv \mathtt{inj}_i^{+[\kappa_1,\ldots,\kappa_i,\ldots,\kappa_n]}\, c' : +[\kappa_1, \ldots, \kappa_i, \ldots, \kappa_n] \\ \Psi;\Delta;\Gamma \vdash e[c'/\beta] : \tau \ \mathbf{exp}\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{let}_\tau\,\mathtt{inj}_i\,\beta = (c, sv)\,\mathtt{in}\,e : \tau \ \mathbf{exp}}$$

**Well-formed** *hval* $\boxed{\Psi \vdash hval : \tau \ \mathbf{hval}}$

$$\frac{\begin{array}{c}\bullet \vdash \kappa_i \ \mathbf{ok} \quad (i \in 1 \ldots k) \\ \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k \vdash \tau_i : \mathrm{T}_{32} \quad (i \in 1 \ldots m) \\ \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k \vdash \phi_i : \mathrm{T}_{64} \quad (i \in 1 \ldots n) \\ \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k \vdash \tau : \mathrm{T}_{32} \\ \Psi; \alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k; x_1{:}\tau_1, \ldots, x_m{:}\tau_m, z_1{:}\phi_1, \ldots, z_n{:}\phi_n \vdash e : \tau \ \mathbf{exp}\end{array}}{\begin{array}{l}\Psi \vdash \mathtt{code}_\tau[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k](x_1{:}\tau_1, \ldots, x_m{:}\tau_m)(z_1{:}\phi_1, \ldots, z_n{:}\phi_n).e : \\ \qquad \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_k{:}\kappa_k]\,\mathtt{Code}(\tau_1, \ldots, \tau_m)(\phi_1, \ldots, \phi_n)(\tau) \ \mathbf{hval}\end{array}}$$

**Well-formed heap** $\boxed{\Psi \vdash d \ \mathbf{ok}}$

$$\frac{\vdash \Psi \ \mathbf{ok}}{\Psi \vdash \epsilon \ \mathbf{ok}}$$

$$\frac{\begin{array}{c}\Psi[\ell{:}\tau] \vdash hval : \tau \ \mathbf{hval} \\ \Psi[\ell{:}\tau] \vdash d \ \mathbf{ok}\end{array}}{\Psi[\ell{:}\tau] \vdash d, \ell{:}\tau \mapsto hval \ \mathbf{ok}}$$

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

$$\begin{array}{ll}\Psi(\epsilon) & \stackrel{\mathrm{def}}{=} \bullet \\ \Psi(d, \ell{:}\tau \mapsto hval) & \stackrel{\mathrm{def}}{=} \Psi(d), \ell{:}\tau\end{array}$$

$$\frac{\Psi, \Psi(d) \vdash d \ \mathbf{ok} \quad \Psi, \Psi(d); \bullet; \bullet \vdash e : \tau \ \mathbf{exp}}{\Psi \vdash \texttt{letrec}\, d \,\texttt{in}\, e : \tau}$$

# Appendix C

# TILTAL static and dynamic semantics

## C.1 TILTAL 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** $\boxed{\Delta \vdash \kappa \ \mathbf{ok}}$

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

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

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

$$\frac{\Delta \vdash \Gamma \ \mathbf{ok}}{\Delta \vdash \Gamma \to 0 : \mathrm{T}_{32}}$$

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

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

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

$$\frac{\Delta \vdash c \equiv \bigvee(\tau_0 :: \cdots :: \tau_i :: \cdots :: \mathtt{nil}) : \mathrm{T}_{32}\,\mathtt{list}}{\Delta \vdash \mathtt{inj\_union}_{(i,c)} : \tau_i \Rightarrow c}$$

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

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

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

$$\frac{\vdash \Delta \ \mathbf{ok}}{\Psi; \Delta \vdash \epsilon : \epsilon} \qquad \frac{\begin{array}{c}\Psi; \Delta \vdash w : \tau \\ \Psi; \Delta \vdash s : \sigma\end{array}}{\Psi; \Delta \vdash w \triangleright_{32} s : \tau \triangleright_{32} \sigma} \qquad \frac{\begin{array}{c}\Psi; \Delta \vdash l : \phi \\ \Psi; \Delta \vdash s : \sigma\end{array}}{\Psi; \Delta \vdash l \triangleright_{32} s : \phi \triangleright_{64} \sigma}$$

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

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

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

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

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

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

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

**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** $\boxed{\Psi; \Delta; \Gamma \vdash i \Rightarrow \Gamma}$

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma(wreg) = \texttt{Dyn} \\ \Psi;\Delta;\Gamma \vdash sv_1 : \texttt{Dyntag}(\tau) \quad \Psi;\Delta;\Gamma \vdash sv_2 : \Gamma\{\mathbf{r}{:}\times[\texttt{Dyntag}(\tau),\tau]\} \to 0\end{array}}{\Psi;\Delta;\Gamma \vdash \texttt{brdyn}\,\mathbf{r}, sv_1, sv_2 \Rightarrow \Gamma}$$

$$\frac{\vdash \Psi\ \mathbf{ok} \quad \Delta \vdash \tau : \mathrm{T}_{32}}{\Psi;\Delta;\Gamma \vdash \texttt{dyntag}_\tau\,\mathbf{r} \Rightarrow \Gamma\{\mathbf{r}{:}\,\texttt{Dyntag}(\tau)\}}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**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** $\boxed{\vdash(H,R,I)\ \textbf{ok}}$

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

## C.2 TILTAL dynamic semantics

### C.2.1 Definitions

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

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

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

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

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

### C.2.2 Transitions

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

**Figure C.1: TILTAL** transitions (part I)

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

**Figure C.2: TILTAL** transitions (part II)

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

**Figure C.3: TILTAL** transitions (part III)

# Bibliography


[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.

[Cra03] Karl Crary. Toward a foundational typed assembly language. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 198–212, New York, NY, USA, 2003. ACM Press.

[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.

[DHB90] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-driven code generation. Technical Report 302, Indiana University Computer Science Department, Bloomington, IN, February 1990.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247. ACM Press, 1993.

[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.

[MCGW02] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[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*, 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.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[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.

[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.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 313–323, New York, NY, USA, 1998. ACM Press.

[THS98] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

[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.

[VC03] Joseph C. Vanderwaart and Karl Crary. A typed interface for garbage collection. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 109–122, New York, NY, USA, 2003. ACM Press.

[VDP+03] Joseph C. Vanderwaart, Derek R. Dreyer, Leaf Petersen, Karl Crary, and Robert Harper. Typed compilation of recursive datatypes. In *Proceedings of the TLDI 2003: ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 98–108, New Orleans, LA, January 2003.

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071, 2001.