An Expressive, Scalable Type Theory for Certified Code

Karl Crary Joseph C. Vanderwaart

Carnegie Mellon University

Abstract

We present the type theory LTT, intended to form a basis for typed target languages, providing an internal notion of logical proposition and proof. The inclusion of explicit proofs allows the type system to guarantee properties that would otherwise be incompatible with decidable type checking. LTT also provides linear facilities for tracking ephemeral properties that hold only for certain program states.

Our type theory allows for re-use of typechecking software by casting a variety of type systems within a single language. We illustrate our methodology of representation by means of two examples, one functional and one stateful, and describe the associated operational semantics and proofs of type safety.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Miscellaneous;
F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs.

General Terms

Languages, Security

1 Introduction

Certified code is a general strategy for providing safety assurances to extensible systems without utilizing hardware-based protection mechanisms. In a certified code architecture, the supplier of any extension code accompanies his or her code with some sort of checkable evidence that the code is safe to execute. Then the consumer of that extension code verifies that safety evidence, thereby establishing the code's safety.

A variety of certified code architectures exist, differing in the kind of code that is certified and in the form that the safety evidence takes. For example, the SPIN system [2] certifies source-level Modula-3 programs, and the Java Virtual Machine architecture [13] certifies intermediate-level, bytecode programs. Recently, there has been considerable interest in certified code architectures that operate at the level of executables, thereby eliminating the need for the code consumer either to trust the correctness of a just-in-time compiler, or to incur the performance cost of an interpreter.

Two main directions have been explored for executable-level certified code: the *proof-oriented* approach exemplified by Proof-Carrying Code (PCC) [17, 1], and the *type-theoretic* approach exemplified by Typed Assembly Language (TAL) [16]. In the proof-oriented approach, executable programs are accompanied by explicit proofs of safety expressed in a formal logic. Safety is then verified by checking the correctness of the proof. In the type-theoretic approach, executable programs are presented in a strongly typed executable language, for which a type safety theorem ensures safety. Safety is then verified by typechecking.

In fact, the two approaches work out to be more similar than this might suggest, as the proof-oriented approach tends to be rather type-theoretic in practice. PCC safety proofs are usually structured using types, and existing implementations of PCC all use the Edinburgh Logical Framework (LF) [8] as their formal logic, in which proof checking boils down to type checking. Nevertheless, there is an important difference between the extrinsic safety evidence of the proof-oriented approach, and the intrinsic safety evidence of the type-theoretic approach.

Although the proof-oriented and type-theoretic approaches each have various strengths and weaknesses we will not discuss here, the proof-oriented approach has had two main advantages not enjoyed by the type-theoretic approach:

First, the proof-oriented approach has been able to provide greater expressive power than has the type-theoretic approach. This has been a direct result of the insistence in TAL on tractable typechecking, which has limited the allowable complexity of the type system. In contrast, in PCC safety arguments can (in principle) be arbitrarily complex, because they are backed up by explicit proofs that the code consumer need not be able to reproduce.

Second, although both approaches have been shown to be scalable to more powerful type systems [15, 26, 22, 1, 4], the proof-oriented approach enjoys much greater *internal* scalability in the following sense: Each new extension to TAL requires a new type system, and therefore requires a new typechecker and a new type safety proof. In contrast, extensions to PCC do not change the logic in which proofs are expressed, so the proof checker need not change. Thus, PCC has been scalable from within the system, while TAL has been scaled from without.

In this paper we present a type theory devised to incorporate these advantages of the proof-oriented approach. Our type theory contains an explicit notion of logical propo-

This material is based on work supported in part by NSF grants CCR-9984812 and CCR-0121633. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

ICFP'02, October 4–6, 2002, Pittsburgh, Pennsylvania, USA.

sition and proof (encoded using the judgments-as-types correspondence) and allows such propositions to stand in function domains and codomains, thereby allowing arbitrarily complicated preconditions and postconditions. This introduction of propositions and proofs as citizens of the type theory provides a natural and direct solution to the expressiveness problem.

Although our ultimate interest is in certified code, in this paper we present a high-level core language; this allows us to abstract from the idiosyncrasies of Typed Assembly Language that are not pertinent our present purposes. Our type theory, called LTT (for "logical type theory"), is structured as a (higher-order) polymorphic lambda calculus, augmented with the constructs of the LF type theory. We chose LF for the logical fragment of our language because it was specifically designed for encoding type systems; it has been used very effectively in certified code already; excellent tools exist for constructing, manipulating and checking LF proofs; and it fits very nicely into our type theory. LTT is designed in such a way that existing results and tools pertaining to LF can be taken off-the-shelf, without any substantial modification.

Just as LF is instantiated to various different logics by the choice of a signature (which provide kinds for types constants and types for term constants), LTT is instantiated to different type theories by the choice of a signature, specifying ordinary types and terms as well as proof types and terms. This provides internal scalability, as scaling LTT requires only changes to the signature, not changes to the typechecker.

Adding the logical power of LF alone provides substantial expressive power for extensions to the type theory. As an example, we show how LTT can, by appropriate choice of a signature, express a type system for arrays without automatic bounds checking, following the ideas of Xi, Pfenning, and Harper [28, 27]. In this example, a well-typed array subscript operation must be supplied with a proof that the subscript is within the appropriate bounds. This example is typical of functional extensions.

In the intuitionistic type theory of LF, once a fact holds, it holds forever. Accordingly, there is never a need for proofs to go away. This is satisfactory for purely functional programming, and it also suffices for some stateful type systems as well. For example, the references of Standard ML [14], once created, never disappear. However, we can provide considerably more expressive power for stateful programming by going beyond just intuitionistic proofs and adding linear proof constructs from Linear LF [3] as well. This allows the proof of facts that hold for the current state, but may later cease to hold; any operation that may falsify such a fact will arrange to consume that fact's proof. As an example, we show how to support revocable capabilities in the style of Walker *et al.*'s region type system [26].

This paper is organized as follows: We begin in Section 2 by presenting the intuitionistic fragment of LTT. In Section 3 we give a sample application of LTT to arrays. In Section 4 we present the full language for linear, stateful programming and show its application to revocable capabilities. In Section 5 we give an example of the application of linearity to stateful programming. These sections are primarily interested in the static semantics of LTT and treat its operational semantics only informally. We then illustrate the method of defining operational semantics for instances of LTT in Section 6.

The intuitionistic fragment of LTT is similar in spirit to recent work by Shao *et al.* [21], which was developed independently. In Section 7 we compare the two systems and discuss the methodological differences in their use.

This paper assumes familiarity with linear logic [7, 24] and with the propositions-as-types correspondence [11]. Additional familiarity with LF and logical frameworks in general will be helpful, but is not required.

proof kinds	K	$::=$	$P \mid \Pi u{:}A.K$
families	A	$::=$	$a \mid Ak \mid \lambda u{:}A_1.A_2 \mid A\,M \mid \Pi u{:}A_1.A_2$
objects	M	$::=$	$u \mid Mk \mid \lambda u{:}A.M \mid M_1 M_2$

Figure 1: Proof Language Syntax

kinds	k	$::=$	$T \mid k_1 \to k_2 \mid \Pi a{:}K.k \mid \Pi u{:}A.k$
con's	c	$::=$	$\alpha \mid ck \mid$ $\lambda\alpha{:}k.c \mid c_1 c_2 \mid \lambda a{:}K.c \mid c\,A \mid \lambda u{:}A.c \mid$ $c\,M \mid c_1 \to c_2 \mid c_1 \times c_2 \mid$ $\Pi\alpha{:}k.c \mid \Sigma\alpha{:}k.c \mid \Pi a{:}K.c \mid \Sigma a{:}K.c \mid$ $\Pi u{:}A.c \mid \Sigma u{:}A.c$
terms	e	$::=$	$x \mid ek \mid$ $\lambda x{:}c.e \mid e_1 e_2 \mid \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e \mid$ $\lambda\alpha{:}k.e \mid e\,c \mid \mathsf{pack}\,\langle c, e\rangle\ \mathsf{as}\ \Sigma\alpha{:}k.c \mid$ $\mathsf{let}\ \langle\alpha, x\rangle = e_1\ \mathsf{in}\ e_2 \mid$ $\lambda a{:}K.e \mid e\,A \mid \mathsf{pack}\,\langle A, e\rangle\ \mathsf{as}\ \Sigma a{:}K.c \mid$ $\mathsf{let}\ \langle a, x\rangle = e_1\ \mathsf{in}\ e_2 \mid$ $\lambda u{:}A.e \mid e\,M \mid \mathsf{pack}\,\langle M, e\rangle\ \mathsf{as}\ \Sigma u{:}A.c \mid$ $\mathsf{let}\ \langle u, x\rangle = e_1\ \mathsf{in}\ e_2$
contexts	Γ	$::=$	$\epsilon \mid \Gamma, \alpha{:}k \mid \Gamma, x{:}c \mid \Gamma, a{:}K \mid \Gamma, u{:}A$

Figure 2: Programming Language Syntax

2 Intuitionistic LTT

The LTT type theory consists of two parts: a proof sublanguage, and a computational programming language built around it. LTT is structured with a syntactic division between the proof language and the surrounding programming language, allowing the proof language to be precisely the LF type theory. This design allows us to reuse a considerable body of existing LF results—particularly metatheoretic proofs—and tools. The surrounding programming language is influenced by the proof language, but not vice versa.

2.1 The Proof Language

The proof language (given in Figure 1) consists of three syntactic classes: objects (M), which are the terms of the proof language; families (A), which are the types and type constructors; and proof kinds (K), which are the "types" of families. As in LF (and as we illustrate by example below), objects implement individuals and formulas of the logic, inference rules, and complete proofs.

Families of kind P are the types of objects; these specify classes of individuals and logical assertions. Families of a higher kind $\Pi u{:}A.K$ are functions mapping objects in family A to families in kind K, where u stands for the argument and may appear free in K. The family constructs are variables a,

constants Ak, lambda abstractions $\lambda u{:}A_1.A_2$, applications of functions $A\,M$, and dependent function spaces $\Pi u{:}A_1.A_2$ (where u again stands for the argument and is permitted to appear free in A_2). When u does not appear free in A_2, we will often write $\Pi u{:}A_1.A_2$ as $A_1 \to A_2$, and similarly for kinds. The object constructs are variables u, constants Mk, lambda abstractions $\lambda u{:}A.M$, and function applications $M_1 M_2$.

Constants are drawn from an unspecified infinite set. As in LF, the kinds of family constants and the families of object constants are given by a signature, which is simply a mapping from constants to their kinds or families. Unlike LF, we permit signatures to be infinite (and therefore we do not provide syntactic rules for them), but this causes no complication to the metatheory, as only finitely many constants can appear in any given expression.

Our proof language also differs from the LF type theory in that it includes family variables (which LF omits). This difference also does not complicate the metatheory of the proof language since it provides no binding construct for family variables, and therefore the proof language may consider variables simply to be additional constants. However, family variables will be useful in the full language.

Example To illustrate the use of the proof language, we provide the following example (abbreviated from Harper *et al.* [8]) of a fragment of first-order logic. Suppose we wish to reason about natural numbers t given by the syntax

$$t ::= u \mid 0 \mid \mathit{succ}(t)$$

and first-order propositions over the natural numbers given by the syntax:

$$\varphi ::= t_1 = t_2 \mid \varphi_1 \supset \varphi_2 \mid \forall u.\varphi$$

We first implement the syntax by introducing into the signature the constants:

i	:	P	o	:	P
$\mathtt{z}$	:	i	$\mathtt{eq}$	:	$i \to i \to o$
$\mathtt{s}$	:	$i \to i$	$\mathtt{impl}$	:	$o \to o \to o$
			$\mathtt{all}$	:	$(i \to o) \to o$

The family i contains the individuals of the logic (the natural numbers in this case), and o contains the propositions. Note that each syntactic form other than variables has its own constant to represent it. In the LF methodology, variables in the logic are always represented simply by variables in the LF type theory; this allows for a very elegant treatment of binding. For example, the proposition $\forall u.\, u = u$ is represented by the object $\mathtt{all}\,(\lambda u{:}i.\,\mathtt{eq}\,u\,u)$.

With syntax taken care of, we next wish to implement judgements and proofs. The relevant judgement in this case is truth of propositions, which is implemented by the constant $tr : o \to P$. For any proposition φ implemented by M, $tr\,M$ will be inhabited exactly when φ is provable. It remains to give proof terms (representing inference rules of the logic) inhabiting the $\mathtt{tr}$ family. A few examples of these are:

$\mathtt{eqrefl}$	:	$\Pi u{:}i.\, tr\,(\mathtt{eq}\,u\,u)$
$\mathtt{implelim}$	:	$\Pi u{:}o.\,\Pi v{:}o.\, tr\,(\mathtt{impl}\,u\,v) \to tr\,u \to tr\,v$
$\mathtt{allintro}$	:	$\Pi f{:}i \to o.\,(\Pi u{:}i.\, tr\,(f\,u)) \to tr\,(\mathtt{all}\,f)$

For example, the judgement that $\forall u.\, u = u$ is true is represented by the family $tr\,(\mathtt{all}\,(\lambda u{:}i.\,\mathtt{eq}\,u\,u))$, and is proved by the object $\mathtt{allintro}\,(\lambda u{:}i.\,\mathtt{eq}\,u\,u)\,(\lambda u{:}i.\,\mathtt{eqrefl}\,i)$. Note that here, unlike the usual propositions-as-types correspondence, judgements are types and propositions are merely terms (not types). A full account of first-order logic in LF appears in Harper *et al.* [8].

Harper *et al.* also give a proof of the *adequacy* property, which states that there is a compositional bijection between derivations in first-order logic and well-formed LF terms (with the respect to this signature) of the appropriate type. A theorem of this form must be proven in order for any representation in LF to be considered successful, and it is essential to the appropriateness of an encoding for use with LTT. It is this adequacy property of a representation that gives derivations encoded in LF the force of actual proofs, so it will play a critical role when we discuss type safety in LTT. We revisit this issue in Section 6.1.

2.2 The Programming Language

The LTT programming language is the higher-order polymorphic lambda calculus augmented with products, and with dependent products and sums over proof kinds and families. The syntax appears in Figure 2 and consists of three classes: terms (e), type constructors (c, usually called "constructors" for short), and kinds (k). Constructors of kind T are the types of terms; we will often use the metavariable τ for constructors intended to be types. Contexts are used to assign a type, kind, or family to each variable.

The types are ordinary functions and products ($\tau_1 \to \tau_2$ and $\tau_1 \times \tau_2$), dependent products over kinds, proof kinds and families ($\Pi\alpha{:}k.\tau$, $\Pi a{:}K.\tau$, and $\Pi u{:}A.\tau$), and dependent sums over the same classes ($\Sigma\alpha{:}k.\tau$, $\Sigma a{:}K.\tau$, and $\Sigma u{:}A.\tau$). When the variable being bound does not appear in the body, we will often write dependent products with an $\to$ and dependent sums with a $\times$. The types $\Pi\alpha{:}k.\tau$ and $\Sigma\alpha{:}k.\tau$ are also often rendered with the quantifiers $\forall$ and $\exists$ in place of Π and Σ, and we will sometimes do so as well.

The higher-kind constructors are lambda abstractions over kinds, proof kinds and families and have the usual elimination forms. Functions abstracting over proof kinds and families have the dependent product kinds $\Pi a{:}K.k$ and $\Pi u{:}A.k$ (which we write using an $\to$ when the variable does not appear free in k). Functions abstracting over kinds have the usual kind (no dependency is necessary since constructors cannot appear within kinds).

At the term level, dependent products are introduced and eliminated using the usual abstraction and application constructs. Dependent sums are introduced using pack expressions (*e.g.,* $\mathsf{pack}\,\langle M, e\rangle\,\mathsf{as}\,\Sigma u{:}A.\tau$) generating the indicated type, and eliminated through pattern matching using let expressions. Functions and products are introduced and eliminated in the usual ways.

As in the proof language, constructor and term constants (ck and ek) are drawn from an unspecified infinite set, and are assigned kinds and types by a signature. Signatures are formalized in Section 2.3.

An extended example of the use of LTT in practice is given in Section 3.

2.3 Static Semantics

To begin defining the static semantics of LTT, we must hammer down the notion of a signature. Signatures, since they may be infinite, are not given by expressions within the type theory. They are defined as follows:

Judgement	Interpretation
$\Gamma \vdash_S k$ kind	k is a valid kind
$\Gamma \vdash_S c : k$	c is a valid constructor of kind k
$\Gamma \vdash_S e : \tau$	e is a valid term of type τ
$\Gamma \vdash_S K$ pkind	K is a valid proof kind
$\Gamma \vdash_S A : K$	A is a valid family of kind K
$\Gamma \vdash_S M : A$	M is a valid object of family A
$\vdash_S \Gamma$ context	Γ is a valid context
$\Gamma \vdash_S k_1 = k_2$ kind	k_1 and k_2 are equal kinds
$\Gamma \vdash_S c_1 = c_2 : k$	c_1 and c_2 are equal constructors
$\Gamma \vdash_S K_1 = K_2$ pkind	K_1 and K_2 are equal proof kinds
$\Gamma \vdash_S A_1 = A_2 : K$	A_1 and A_2 are equal families
$\Gamma \vdash_S M_1 = M_2 : K$	M_1 and M_2 are equal objects

Figure 3: Intuitionistic LTT Judgements

Definition 2.1 An *(intuitionistic) signature* S is a mapping[1] of constructor constants (ck) to kinds, term constants (ek) to constructors, family constants (Ak) to proof kinds, and object constants (Mk) to families, together with a well-founded ordering on the non-term constants in the domain of the mapping. We write $<_S$ for the ordering associated with S.

The judgement forms for LTT's static semantics are given in Figure 3. We follow Harper and Pfenning's treatment of the metatheory of LF [10], using typed equality judgements rather than $\beta\eta$-conversion; equality judgements are required for all classes except terms, which alone cannot appear inside of types. The typing rules for LTT are the expected ones, and are given in Appendix B (for full LTT, including linearity). As usual, we consider alpha-equivalent expressions to be identical. We write the simultaneous capture-avoiding substitution of $E_1, \ldots, E_n$ for $X_1, \ldots, X_n$ in E as $E[E_1 \cdots E_n / X_1 \cdots X_n]$.

All LTT judgements are predicated over a signature, and are to be considered meaningful only when that signature is well-formed. For any (non-term) constant $sk \in Dom(S)$, we write $S \restriction sk$ for the restriction of S to constants less than sk (by $<_S$). We also write $TLP(S)$ for the termless portion of the signature S. Then we can define well-formedness as follows:

Definition 2.2 An intuitionistic signature S is *well-formed* if:

- for all $Ak \in Dom(S)$, $\vdash_{S \restriction Ak} S(Ak)$ pkind, and
- for all $Mk \in Dom(S)$, $\vdash_{S \restriction Mk} S(Mk) : P$, and
- for all $ck \in Dom(S)$, $\vdash_{S \restriction ck} S(ck)$ kind, and
- for all $ek \in Dom(S)$, $\vdash_{TLP(S)} S(ek) : T$.

Typechecking may be shown decidable for intuitionistic LTT, but we will defer discussion of typechecking to Section 4.4 where we discuss it for full LTT. We discuss the operational semantics of LTT and its type safety properties in Section 6.

[1] We define a mapping from B to C to be a function from some subset of B into C, and thus it may be undefined on some members of B.

3 Example: Arrays and Arithmetic

In this section we will develop a detailed example of an LTT signature, namely one that uses a theory of integer arithmetic to eliminate safely the dynamic checking of array bounds.

In order to allow unchecked array accesses safely, we must be able to verify statically that the indices being accessed in array operations are within the appropriate bounds. The reason this is complicated is that the index values are simply integers and may therefore be the results of arbitrarily complex arithmetic, making it impossible in general for a static analyzer to verify that they satisfy the appropriate constraints. Our solution is to use an LF representation of some theory of arithmetic—any sufficiently expressive theory that may be adequately encoded in LF will do—and allow the program itself to construct proofs that the preconditions of array accesses are satisfied. The array operations themselves will be unchecked at run time but will require such proofs as arguments to ensure that the necessary conditions are met when they are called.

The basic families and types required for this task are as follows:

$$\begin{array}{rlrl} o & : P & \mathit{array} & : \mathit{Int} \to T \to T \\ \mathit{tr} & : o \to P & \mathit{int} & : T \\ \mathit{Int} & : P & S_{\mathit{Int}} & : \mathit{Int} \to T \end{array}$$

As in the shorter example given earlier, o is the family of propositions, while tr maps any proposition to the family of proofs that that proposition is true. Int is the family of objects that represent integer formulas in the theory of arithmetic (similar to i in the simple example), and for any integer object N and any type τ, $\mathit{array}\, N\, \tau$ is the type of arrays of size N with elements of type τ. The type int classifies integer values about which nothing is known—integers that are computed based on user input, for example. S_{Int} is used to construct singleton types corresponding to particular objects of family Int; if $N : Int$, then $S_{Int}\, N$ is a type containing exactly one value, namely the integer represented by N.

Our theory of arithmetic is represented in LTT by a collection of several object constants including:

$$\begin{array}{rlrl} \texttt{z} & : \mathit{Int} & \texttt{eq} & : \mathit{Int} \to \mathit{Int} \to o \\ \texttt{s} & : \mathit{Int} \to \mathit{Int} & \texttt{lt} & : \mathit{Int} \to \mathit{Int} \to o \\ \texttt{neg} & : \mathit{Int} \to \mathit{Int} & \texttt{not} & : o \to o \\ \texttt{plus} & : \mathit{Int} \to \mathit{Int} \to \mathit{Int} & \texttt{and} & : o \to o \to o \end{array}$$

$$\begin{array}{l} \texttt{and-i} : \Pi u{:}o.\Pi v{:}o.\mathit{tr}\, u \to \mathit{tr}\, v \to \mathit{tr}\, (\texttt{and}\, u\, v) \\ \texttt{commute} : \Pi u{:}\mathit{Int}.\Pi v{:}\mathit{Int}.\mathit{tr}\, (\texttt{eq}\, (\texttt{plus}\, u\, v)\, (\texttt{plus}\, v\, u)) \end{array}$$

As in the earlier example of our proof language, `z` represents zero, and $(\texttt{s}\, M)$ represents the successor of the integer represented by M. The proposition $\texttt{eq}\, M\, N$ states that the integers represented by M and N are equal, and the proposition $\texttt{lt}\, M\, N$ states that M is less than N. We mention here just two proof constructors (rules of inference): `and-i` corresponds to the and-introduction rule of propositional logic and will appear in an example later; `commute` represents the commutative property of addition. All the other proof-building constants are omitted to save space.

Now that we can express and prove properties of numbers using an LF-like representation of arithmetic, the types of

```
fun checked_aget arr size index =


    if index < 0 then
        raise Sub
    else
        if index < size then

            unchecked_sub(arr,index) (* Safe? *)
        else
            raise Sub
```

$$
\begin{array}{l}
\mathit{checked\_aget} = \\
\lambda u{:}\mathit{Int}.\lambda \mathit{arr}{:}(\mathit{array}\,u\,\mathit{int}).\lambda \mathit{size}{:}(S_{\mathit{Int}}\,u).\lambda \mathit{index}{:}\mathit{int}. \\
\mathsf{let}\ \langle v, \mathit{index}'\rangle = \mathtt{focus}\ \mathit{index}\ \mathsf{in} \\
\quad \mathtt{test}\ \mathit{int}\ (\mathtt{lt}\,v\,\mathtt{z})\ (\mathtt{<}\,v\,\mathtt{z}\,\mathit{index}'\,\overline{0}) \\
\qquad (\lambda p{:}(\mathit{tr}\ (\mathtt{lt}\,v\,\mathtt{z})).\mathtt{raise\ Sub}) \\
\qquad (\lambda p{:}(\mathit{tr}\ (\mathtt{not}\ (\mathtt{lt}\,v\,\mathtt{z}))). \\
\qquad\quad \mathtt{test}\ \mathit{int}\ (\mathtt{lt}\,v\,u)\ (\mathtt{<}\,v\,u\,\mathit{index}'\,\mathit{size}) \\
\qquad\qquad (\lambda q{:}(\mathit{tr}\ (\mathtt{lt}\,v\,u)). \\
\qquad\qquad\quad \mathtt{aget}\ \mathit{int}\,u\,v\ (\mathtt{and\text{-}i}\ldots p\,q)\ \mathit{arr}\ \mathit{index}') \\
\qquad\qquad (\lambda q{:}(\mathit{tr}\ (\mathtt{not}\ (\mathtt{lt}\,v\,u))). \\
\qquad\qquad\quad \mathtt{raise\ Sub}))
\end{array}
$$

Figure 4: Checked subscript in SML and LTT

the array operations are not surprising:

$$
\begin{array}{rl}
\mathtt{mkarray} : & \Pi\alpha{:}T.\,\Pi u{:}\mathit{Int}. \\
& \quad \mathit{tr}\ (\mathtt{not}\ (\mathtt{lt}\,u\,\mathtt{z})) \to S_{\mathit{Int}}\,u \to \alpha \to \mathit{array}\,u\,\alpha \\
\mathtt{aget} : & \Pi\alpha{:}T.\,\Pi u{:}\mathit{Int}.\,\Pi v{:}\mathit{Int}. \\
& \quad \mathit{tr}\ (\mathtt{and}\ (\mathtt{not}\ (\mathtt{lt}\,v\,\mathtt{z}))(\mathtt{lt}\,v\,u)) \to \\
& \quad \mathit{array}\,u\,\alpha \to S_{\mathit{Int}}\,v \to \alpha \\
\mathtt{aset} : & \Pi\alpha{:}T.\,\Pi u{:}\mathit{Int}.\,\Pi v{:}\mathit{Int}. \\
& \quad \mathit{tr}\ (\mathtt{and}\ (\mathtt{not}\ (\mathtt{lt}\,v\,\mathtt{z}))(\mathtt{lt}\,v\,u)) \to \\
& \quad \mathit{array}\,u\,\alpha \to S_{\mathit{Int}}\,v \to \alpha \to \mathit{array}\,u\,\alpha
\end{array}
$$

Each operation enforces its precondition by requiring the client to pass a proof of the appropriate fact before the term-level function may be applied. In particular, the `mkarray` function requires a proof that the size of the array to be created is non-negative, and the `aget` and `aset` functions each require a proof that the index being accessed is at least zero but less than the size of the array.

This is not all we need, however, since we as yet have no way of obtaining any values of type $S_{\mathit{Int}}\,M$ for any integer expression M, and we have no mechanism allowing information the program discovers at run time to play a role in proving that any precondition is satisfied. We will address these two issues one at a time.

First, to link the (static) LF representation of arithmetic with the (dynamic) integer values manipulated at run time, it is necessary to establish a correspondence between the singleton integer types $S_{\mathit{Int}}\,M$ and the type *int*. For this purpose we include the coercions `focus` and `blur`, which define an isomorphism between *int* and $\Sigma u{:}\mathit{Int}.S_{\mathit{Int}}\,u$, the type of a singleton value whose identity is hidden by the weak dependent sum.

$$
\begin{array}{rl}
\mathtt{focus} : & \mathit{int} \to \Sigma u{:}\mathit{Int}.S_{\mathit{Int}}\,u \\
\mathtt{blur} : & (\Sigma u{:}\mathit{Int}.\,S_{\mathit{Int}}\,u) \to \mathit{int}
\end{array}
$$

We must also include singleton-aware arithmetic operations and literal values of singleton type, for example:

$$
\begin{array}{l}
\mathtt{+} : \Pi u{:}\mathit{Int}.\,\Pi v{:}\mathit{Int}.\,S_{\mathit{Int}}\,u \to S_{\mathit{Int}}\,v \to S_{\mathit{Int}}\,(\mathtt{plus}\,u\,v) \\
\overline{n} : S_{\mathit{Int}}\,(\mathtt{s}^n\mathtt{z})
\end{array}
$$

Note that in some cases, the programmer may wish to use "singleton-blind" arithmetic operations such as $\mathtt{+}' : \mathit{int} \to \mathit{int} \to \mathit{int}$; these can all be defined using the singleton-aware operations, with the help of `focus` and `blur`.

Now we address the problem of incorporating information discovered by the program at run time into our proof system. Intuitively, we would like a program to be able to use the outcome of a test as evidence in a proof: in the then-branch of a conditional it should be possible to assume the guard condition to be true, and in the else-branch it should be possible to assume it to be false. To accomplish this, we replace ordinary boolean values with special ones whose types associate them with the proposition whose truth, or falsehood, they assert. We provide singleton-aware integer comparison operations to introduce these special values, and eliminate them with a special conditional construct that allows each of its branches to assume the truth of a proposition describing the outcome of the test. The constants necessary to do this are as follows:

$$
\begin{array}{rcl}
S_o & : & o \to T \\
\mathtt{=} & : & \Pi u{:}\mathit{Int}.\,\Pi v{:}\mathit{Int}.\,S_{\mathit{Int}}\,u \to S_{\mathit{Int}}\,v \to S_o\,(\mathtt{eq}\,u\,v) \\
\mathtt{<} & : & \Pi u{:}\mathit{Int}.\,\Pi v{:}\mathit{Int}.\,S_{\mathit{Int}}\,u \to S_{\mathit{Int}}\,v \to S_o\,(\mathtt{lt}\,u\,v) \\
\mathtt{test} & : & \Pi\alpha{:}T.\,\Pi u{:}o.\,S_o\,u \to (\mathit{tr}\,u \to \alpha) \to \\
 & & \quad (\mathit{tr}\,(\mathtt{not}\,u) \to \alpha) \to \alpha
\end{array}
$$

The constant S_o is used to construct special boolean types: if P is a proposition, then at run time the type $S_o\,P$ will effectively contain only `true` if P is true, and only `false` if P is false. Notice that the type of the conditional operator `test` is reminiscent of an encoding of sum types in polymorphic lambda-calculus; in fact, if LTT had a disjoint union type we could implement the type $S_o\,P$ as $(\mathit{tr}\,P) + (\mathit{tr}\,(\mathtt{not}\,P))$, leaving only `=` and `<` as primitive constants.

We illustrate this treatment of arithmetic with the example shown in Figure 4. On the left is the implementation of a checked array subscript function in Standard ML; on the right is the equivalent LTT function. Both versions perform two tests on the given index value to ensure that the unchecked subscript operation will not fail; in LTT, however, the proof objects make it statically clear that the resulting code is safe. For clarity, we will assume the existence of an SML-like exception mechanism in LTT and will raise the exception `Sub` if the index is out of range.

The code on the right is well-typed, and it is therefore safe to call this function with any integer index. Although initially nothing is known about the value of this parameter, the `focus` coercion attaches an object-level name to it. The use of `test` with the singleton-aware comparison operator allows the safe branch of the code to prove that, if it is reached, the subscript operation will be given valid arguments.

4 Linearity and State

Thus far, we have concerned ourselves with proofs of persistent facts—ones that, once proved, remain true. For exam-

families	A	::=	$\cdots \mid A_1 \multimap A_2 \mid A_1 \,\&\, A_2 \mid \top$
objects	M	::=	$\cdots \mid Mlk \mid \hat{\lambda} u{:}A.M \mid M_1 \hat{} M_2 \mid \langle M_1, M_2 \rangle \mid \pi_1 M \mid \pi_2 M \mid \langle\rangle$
linear contexts	Δ	::=	$\epsilon \mid \Delta, u \hat{:} A$

Figure 5: Linear Proof Language Syntax

ple, if a given integer lies between two other given integers, it will not later find itself outside that range. This is satisfactory for stateless programming, but if we introduce state into our language, we will also find it profitable to deal with ephemeral facts—ones that hold for the current state but perhaps not for some other state.

Proof of ephemeral facts is incompatible with intuitionistic LF; once a proof is constructed, there is no way to make it go away. To add support for ephemeral facts, we introduce linearity into the proof language. With linearity at our disposal, we can arrange for any ephemeral facts on which we rely to appear as linear assumptions, and craft stateful operations to ensure that they consume the assumptions corresponding to any ephemeral facts that they invalidate.

4.1 Linear Proofs

In keeping with our design so far, we use the type theory of Linear LF [3] as our proof language in the presence of state. To obtain Linear LF, we add the additional constructs in Figure 5 to the proof language.

The principal difference in Linear LF is the existence of linear variables. Linear variables represent scarce resources that must be used exactly once. We do not distinguish between linear and intuitionistic variables syntactically; instead, any object variable bound by a linear context or a linear abstraction (discussed below) is considered linear, while an object variable bound by an intuitionistic construct (such as an ordinary context) is considered intuitionistic. Intuitionistic variables are treated the same as variables in LF; in particular, they may be duplicated or discarded. For convenience, we do not distinguish between linear contexts that differ only in the order variables are declared. We also provide linear object constants (Mlk), whose types are given by a linear signature; these too must be used exactly once. We sometimes refer to linear variables and linear object constants jointly as linear resources.

Linear LF provides three new types: linear functions ($A_1 \multimap A_2$), "with" (a.k.a. alternative or additive conjunction, $A_1 \& A_2$), and top ($\top$). Linear functions are introduced by a linear abstraction form ($\hat{\lambda} u{:}A.M$) and eliminated by a linear application form ($M_1 \hat{} M_2$). The defining characteristic of a linear function is that it is guaranteed to use its argument exactly once. Thus, linear functions may be applied to objects containing linear resources; ordinary functions make no such guarantee, so they may not be applied to objects containing linear resources.

With types are introduced by pairs ($\langle M_1, M_2 \rangle$) and eliminated by projection ($\pi_i M$). The defining characteristic of additive conjunction is that each component consumes all available resources. Consequently, $A_1 \& A_2$ provides a choice of either A_1 or A_2 but not both. The choice is determined by which projection operation is used. Top ($\top$) is the unit for with; it has the introduction form $\langle\rangle$ and no elimination form. When $\langle\rangle$ is used, it consumes all available resources. This is useful in some circumstances for collecting unused resources.

At first glance it may appear that the Linear LF type theory allows intuitionistic function types to be dependent, but does not allow linear function types this generality. The reason for this difference is simply that it does not make sense for linear variables to appear in families, since allowing this sort of thing would create a theory in which the expression denoting a type or kind might contain expendable resources and therefore need to appear exactly once.

Two types common in linear logic but not provided in the Linear LF type theory are tensor (a.k.a. simultaneous or multiplicative conjunction, $\otimes$) and "of course" (!). Tensor is provided in LTT in a restricted way, which we discuss below in Section 4.3. "Of course," which indicates an intuitionistic (and therefore duplicatable) object, is not provided; however, as Cervesato and Pfenning point out [3], the intuitionistic function type $A \to B$ inherited from the ordinary LF type theory behaves similarly to the type $(!A \multimap B)$ that can be expressed in logics containing "of course." The argument to such a function must be an object that is well-typed in an empty linear context, which ensures that it does not depend on any linear assumptions.

4.2 Linearity in Programming

The way in which linear constructs in the proof language are intended to represent state in programs may be clarified by examining how these new constructs are made to interact with the programming language. Our interpretation is based on the notion that at any point in the execution of a program, there is a certain set of resources available; these resources might be actual things, like memory locations, that are available for use by the program, or they may be "facts" about the machine state of which the program may take advantage. What is important is that some kinds of program actions may create new resources, such as by allocating some new data structure, or by causing a fact to hold that did not hold before, and other actions may destroy a resource, such as by destroying the data object it refers to, or by invalidating a "fact".

The new judgement forms in linear LTT are listed in Figure 8. To reflect the notion of available resources in the type system, the judgements for objects and terms now include a *linear signature* R and a linear context Δ in addition to the intuitionistic signature and context. Together, these new additions represent the set of resources available for use by the term(s) on the right-hand side of the turnstile; further, each item in their combined domains must each be used exactly once. (The linear signature thus resembles a linear context more than it does an intuitionistic signature, as it must be divided among subterms to ensure proper resource use.) Our intuition is that evaluation of any program term that contains a linear constant or a free linear variable "consumes" the associated resource; thus we will often speak of a term that is well-typed in an empty linear context and with $R = \emptyset$ as "consuming no resources."

It is clear that in order to be consistent with this understanding of resources, syntactic values in the programming language must never contain any linear constants or free linear variables: values themselves do not perform any computation and hence cannot consume anything. Thus, our typing rules must force all values to be typed in an empty

$$\begin{array}{llll} \textit{constructors} & c & ::= & \cdots \mid A \multimap c \\ \textit{terms} & e & ::= & \cdots \mid \hat{\lambda} u{:}A.e \mid e \hat{} M \end{array}$$

Figure 6: Linear Programming Language Syntax

linear context and with an empty linear signature, for example:

$$\frac{\Gamma \vdash_S \tau_1 : T \quad \Gamma, x{:}\tau_1; \epsilon \vdash_{S,\emptyset} e : \tau_2}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}$$

On the other hand, because values never contain linear resources, they may always be freely replicated without violating the linear typing discipline. Since our intended operational semantics is call-by-value, this means that the argument in a function application *may* contain resources: even though the function's body may refer to its formal parameter any number of times, the term being replicated is not the argument expression itself but that expression's eventual value. Thus we have the following somewhat unusual typing rule for function application:

$$\frac{\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \tau_1 \to \tau_2 \quad \Gamma; \Delta_2 \vdash_{S,R_2} e_2 : \tau_1}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} e_1 e_2 : \tau_2}$$

In addition to these modifications to the existing typing judgements, the programming language is extended to support linearity by adding one additional type. The type $A \multimap \tau$ contains functions that consume the resource A (using it exactly once), and return a value of type τ. This type is introduced and eliminated by similar constructs to those in the proof language, as shown in Figure 6. The typing rule for linear functions requires that the body treat the formal parameter as a linear variable, but is otherwise similar to the rule for ordinary functions over families:

$$\frac{\Gamma \vdash_S A : P \quad \Gamma; u \hat{:} A \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \hat{\lambda} u{:}A.e : A \multimap \tau}$$

Interestingly, the restrictions imposed by our resource interpretation of linear constants and variables often lead stateful programming into a closure-passing, continuation-passing style. Since functions must be closed (with respect to linear variables), they must always take resources as arguments, even when those resources were actually available when the function was created, resulting in a closure-passing style (with respect to resources). (The desire to include multiple resources is what leads to the need for tensor (Section 4.3).) Conversely, since functions can never return resources—to do so would involve returning a value containing resources—any function that wants to provide new resources must provide them as arguments to a continuation function. Fortunately this is no hardship: our intended application, Typed Assembly Language, depends essentially on closure- and continuation-passing style anyway.

Example Suppose A and B are ephemeral propositions, and suppose f is a stateful function that requires A to be true, but then falsifies it and causes B to be true instead. For example, A could say that storage cell 1 is valid, B could say that storage cell 2 is valid, and f could deallocate 1 and allocate 2.

Suppose further that A holds in the current state, and consequently there is a available a linear variable $u \hat{:} A$. Then we can give f the type $(B \multimap \tau) \to A \multimap \tau$ and call it with the code $f\,(\hat{\lambda} v{:}B.e) \hat{} u$, where e is some continuation expression requiring a state satisfying B. The order of the arguments is significant: because of our interpretation of resource consumption and the restrictions on linear variable occurrences, linear arguments are generally forced to be last.

Alternatively, suppose g requires A but does not invalidate it. For example, g might read from storage cell 1. Then we can give g the type $(A \multimap \tau) \to A \multimap \tau$ and call it with the code $g\,(\hat{\lambda} u{:}A.e) \hat{} u$, where e is some continuation expression requiring a state satisfying A.

$$\begin{array}{llll} \textit{proof kinds} & K & ::= & \cdots \mid P^+ \\ \textit{families} & c & ::= & \cdots \mid A_1 \otimes A_2 \mid 1 \\ \textit{objects} & M & ::= & \cdots \mid \langle\!\langle M_1, M_2 \rangle\!\rangle \mid \star \\ \textit{terms} & e & ::= & \cdots \mid \mathsf{let}\ \langle\!\langle u_1, u_2 \rangle\!\rangle = M\ \mathsf{in}\ e \mid \\ & & & \mathsf{let}\ \star = M\ \mathsf{in}\ e \end{array}$$

$$\frac{\Gamma \vdash A_i : P^+}{\Gamma \vdash A_1 \otimes A_2 : P^+} \qquad \frac{}{\Gamma \vdash 1 : P^+} \qquad \frac{\Gamma \vdash A : P}{\Gamma \vdash A : P^+}$$

$$\frac{\Gamma; \Delta_i \vdash M_i : A_i}{\Gamma; (\Delta_1, \Delta_2) \vdash \langle\!\langle M_1, M_2 \rangle\!\rangle : A_1 \otimes A_2} \qquad \frac{}{\Gamma; \epsilon \vdash \star : 1}$$

$$\frac{\Gamma; \Delta_1 \vdash M : A_1 \otimes A_2 \quad \Gamma; (\Delta_2, u_1 \hat{:} A_1, u_2 \hat{:} A_2) \vdash e : \tau}{\Gamma; (\Delta_1, \Delta_2) \vdash \mathsf{let}\ \langle\!\langle u_1, u_2 \rangle\!\rangle = M\ \mathsf{in}\ e : \tau}$$

$$\frac{\Gamma; \Delta_1 \vdash M : 1 \quad \Gamma; \Delta_2 \vdash e : \tau}{\Gamma; (\Delta_1, \Delta_2) \vdash \mathsf{let}\ \star = M\ \mathsf{in}\ e : \tau}$$

Figure 7: Tensor

4.3 Tensor

Linear LF provides alternative conjunction, in which both conjuncts consume all available resources, making one or the other conjunct available to the receiver, but not both. Another useful form of conjunction is tensor (or simultaneous conjunction). In tensor (written $A_1 \otimes A_2$), the available resources are divided between the two conjuncts, making both of them simultaneously available to the receiver.

Unfortunately, Cervesato and Pfenning [3] show that inclusion of tensor would invalidate important metatheoretic properties of LF and Linear LF (such as the existence of canonical forms and all known proofs of decidability of typechecking), and consequently tensor is omitted from Linear LF. Cervesato and Pfenning show that this is no major hardship for logical frameworks, one can always work around the absence of tensor in a systematic way. Sadly, this is not so for LTT.

Since term functions are required to be closed with respect to linear variables, it is impossible for a term function to take multiple linear arguments by currying. A partial solution would be to provide multi-argument linear functions primitively, with a type like $A_1 \otimes \cdots \otimes A_n \multimap \tau$. This turns out to be inadequate because it provides insufficient support for polymorphism.

Consider a higher-order function such as `apply` : $\forall \beta{:}T.\forall \gamma{:}T.(\beta \to \gamma) \times \beta \to \gamma$. In LTT, it is essential to be able to provide linear resources to the argument function,

otherwise that function is crippled for stateful computation. This can be solved using polymorphism over families:

$$\texttt{apply} : \forall a{:}P.\, \forall \beta{:}T.\forall \gamma{:}T.\, (\beta \to a \multimap \gamma) \times \beta \to a \multimap \gamma$$

but this solution only works when a can be instantiated with tensor. A multi-argument function would not suffice here, because it cannot be known how many linear arguments are to be passed to the argument function.

We overcome this problem with a trick: tensor is added with a typing rule that prevents it from interacting with the proof language. For convenience only, we include tensor among the syntactic class of families, but we give it a new proof kind P^+. The typing rules for the other constructs of the proof language do not recognize families in P^+, so as far as they are concerned tensor is ill-formed and might as well not exist. This preserves the properties of Linear LF.

To make use of tensor, we add a construct eliminating it in the *programming* part of the language. The construct $\mathsf{let}\ \langle\langle u_1, u_2\rangle\rangle = M\ \mathsf{in}\ e$ decomposes an object $M : A_1 \otimes A_2$ and adds new variables $u_1 \hat{:} A_1$ and $u_2 \hat{:} A_2$ to the linear context. Tensor objects are introduced by $\langle\langle M_1, M_2\rangle\rangle$. We also include a type 1, the unit for tensor, which is eliminated by a similar term-level construct and introduced by $\star$. In contrast to $\langle\rangle$, which consumes all resources, $\star$ consumes no resources. These developments are summarized in Figure 7 (with signatures omitted for brevity).

The `apply` function given above is certainly a trivial example of a higher-order function. Another complication arises when a function passed as an argument might have to be applied more than once, as in the familiar function $\texttt{map} : \forall \beta, \gamma{:}T.\, (\beta \to \gamma) \to \texttt{list}\,\beta \to \texttt{list}\,\gamma$. While it makes sense for the argument of `map` to require some resources to be present, in order to for `map` to be able to apply that function more than once, `map` must require that those resources not be consumed. As discussed at the end of the previous section, expressing the presence of resources in the "post-condition" of a function requires that programs be written in continuation-passing style. Specifically, if the argument to `map` is intended to have the type $\beta \to \gamma$, and if the variable a stands for the family of the resources it needs, then that argument must have the type $\forall \delta{:}T.\, \beta \to (\gamma \to a \multimap \delta) \to a \multimap \delta$ in continuation-passing style. Using a similar transformation, and quantifying over the resources required by the argument function, we obtain:

$$\begin{aligned}\texttt{map} : \forall a{:}P^+.\, \forall \beta, \gamma, \varepsilon{:}T.\ & \\ & (\forall \delta{:}T.\, \beta \to (\gamma \to a \multimap \delta) \to a \multimap \delta) \to \\ & \texttt{list}\,\beta \to (\texttt{list}\,\gamma \to a \multimap \varepsilon) \to a \multimap \varepsilon\end{aligned}$$

4.4 Static Semantics

The principal change to the static semantics of LTT with the addition of linearity is the addition of linear contexts to the judgement forms for terms, objects, and object equality. No linear context is required for the other judgements since, as discussed above, types, families, kinds, and proof kinds are not permitted to contain resources. The new judgements are shown in Figure 8. These three judgements also add a linear signature R. A linear signature assigns families to linear object constants, and is treated similarly to the linear context in that it is divided among subterms to ensure that each linear object constant is used exactly once.

Definition 4.1 A *linear signature* R, is a mapping of linear object constants (*Mlk*) to families. A linear signature R is well-formed (relative to an intuitionistic signature S) if for all $Mlk \in Dom(R)$, $\vdash_S R(Mlk) : P$.

Judgement	Interpretation
$\Gamma; \Delta \vdash_{S,R} e : \tau$	e is a valid term of type τ
$\Gamma; \Delta \vdash_{S,R} M : A$	M is a valid object of family A
$\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : K$	M_1 and M_2 are equal objects
$\Gamma \vdash_S \Delta\ \mathsf{context}$	Δ is a valid linear context

Figure 8: Linear LTT Judgements

The full typechecking rules for LTT appear in Appendix B. The version of Linear LF contained within LTT differs from that of Cervesato and Pfenning in two significant ways. Cervesato and Pfenning's version of Linear LF requires that all objects and families be written in canonical form whereas ours has no such restriction. Also, Cervesato and Pfenning omit the lambda abstraction construct at the family level. Those restrictions were required for their proof of decidable typechecking for Linear LF, and were rarely a real burden in practical use. However, they do substantially complicate the presentation of the type system, so we have removed them here. Accordingly, decidability of LTT typechecking is based on a new proof [23] of decidable typechecking for Linear LF.

As is often the case, typechecking in LTT boils down to the problem of deciding equivalence of types. In LTT this proves to be easy, provided it is possible to decide equivalence of families and objects (apart from which, LTT type constructors are just terms of the simply typed lambda calculus).

Theorem 4.2 *Suppose S and R are well-formed, $\vdash_S \Gamma\ \mathsf{context}$, and $\Gamma \vdash_S \Delta\ \mathsf{context}$. Then it is decidable whether or not $\Gamma \vdash_S A_1 = A_2 : K$ is derivable, and whether or not $\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A$ is derivable.*

The proof [23] is based on a logical relations argument modeled after the analogous proof of Harper and Pfenning [10] for intuitionistic LF. From this it is easy to show decidability of LTT typechecking:

Corollary 4.3 *Suppose S and R are well-formed, $\vdash_S \Gamma\ \mathsf{context}$, and $\Gamma \vdash_S \Delta\ \mathsf{context}$. Then it is decidable whether or not $\Gamma; \Delta \vdash_{S,R} e : \tau$.*

5 Example: Memory Management

In this section we present a strategy for using the linear constructs of LTT to implement safe, explicit allocation and deallocation of memory. The signature we will give here is essentially a simple fragment of the capability calculus of Walker *et al.* [26].

Our simplified memory management signature provides a type of ephemeral reference cells, which resemble those of ML except that they may be explicitly deallocated. Allocation of a new reference cell will result in the creation of a linear resource witnessing that the cell is *valid*—that is, that it is available to be read from, written to or destroyed. The family and type constants we need are these:

$$\begin{aligned} \mathit{cell} &: P \\ \mathit{valid} &: \mathit{cell} \to P \\ \mathit{ref} &: T \to \mathit{cell} \to T \end{aligned}$$

As these declarations show, reference cells are represented in the proof language by objects of family *cell*; for a cell C, *valid* C is a proof family that we will arrange to be inhabited when and only when there exists an actual cell associated with C. The storage cell itself will be a term-level value of type *ref* $\tau\, C$, where τ is the type of the cell's contents.

The operations available on reference cells have the following types:

$$\begin{aligned}
\texttt{new} &: \forall a{:}P^+.\, \forall \beta{:}T.\, \forall \gamma{:}T. \\
&\quad \beta \to (\Pi c{:}\mathit{cell}.\ \mathit{ref}\ \beta\, c \to (\mathit{valid}\, c \otimes a) \multimap \gamma) \to \\
&\quad a \multimap \gamma \\
\texttt{deref} &: \forall a{:}P^+.\, \Pi c{:}\mathit{cell}.\, \forall \beta{:}T.\, \forall \gamma{:}T. \\
&\quad \mathit{ref}\ \beta\, c \to (\beta \to (\mathit{valid}\, c \otimes a) \multimap \gamma) \to \\
&\quad (\mathit{valid}\, c \otimes a) \multimap \gamma \\
\texttt{assign} &: \forall a{:}P^+.\, \Pi c{:}\mathit{cell}.\, \forall \beta{:}T.\, \forall \gamma{:}T. \\
&\quad \mathit{ref}\ \beta\, c \to \beta \to ((\mathit{valid}\, c \otimes a) \multimap \gamma) \to \\
&\quad (\mathit{valid}\, c \otimes a) \multimap \gamma \\
\texttt{free} &: \forall a{:}P^+.\, \Pi c{:}\mathit{cell}.\, \forall \beta{:}T.\, \forall \gamma{:}T. \\
&\quad \mathit{ref}\ \beta\, c \to (a \multimap \gamma) \to (\mathit{valid}\, c \otimes a) \multimap \gamma
\end{aligned}$$

As these types reveal, programs that wish to use references must be written in continuation-passing style in order to manage their resources properly. The functions rely on tensor to allow the caller to pass extra resources; these resources are then forwarded to the continuation along with any others the operation makes available. This allows more than one cell to be valid at a time. Since the variable a in the tensor family is universally quantified, these operators may be instantiated and used no matter what irrelevant assumptions the linear context may contain.

The effect of each operation may be guessed by looking at the difference between the family of the resource passed to the function and the family of the resource the function passes on to its continuation. The `new` operation allocates a new cell and creates a new validity resource to pass on; `free` destroys a cell and consumes the corresponding resource. The `deref` and `assign` functions require that the cell they are accessing be valid, but they do not cause it to become invalid, and so the resource asserting its usability is passed on untouched to the continuation. The result of all this is that at any point in the program, the linear context will contain validity resources in one-to-one correspondence with the cells that are available to be accessed.

6 Operational Semantics

Defining an operational semantics for LTT presents a significant difficulty, namely the treatment of term-level constants. Logical frameworks such as LF and LLF are content to leave the constants in their signatures "uninterpreted"—in fact, the presence of uninterpreted function symbols is the very basis of their ability to represent formulas and proofs. In the programming language part of LTT, however, it is our intention that term-level constants represent primitive values (such as $\overline{n}$), operations (such as `+`) and programming constructs (such as `test`), so they cannot meaningfully be left uninterpreted.

It is almost certainly impossible to give a single operational semantics for LTT that produces the correct behavior for every programming construct the language may ever be used to represent. Instead, a separate set of rules must be designed for each LTT signature, and any desired properties of the semantics (including type safety, which we will discuss) must be proved separately as well. We will illustrate our methodology by outlining the operational semantics for the two major examples in this paper: arrays and memory management.

$$\frac{}{\texttt{mkarray}\,\tau\, M\, M'\, \overline{n}\, v \mapsto [\underbrace{v, \ldots, v}_{n\ \mathit{times}}]}\ (n \geq 0)$$

$$\frac{}{\texttt{aget}\, c\, M_1\, M_2\, M_3\, [v_0, \ldots, v_{(n-1)}]\, \overline{i} \mapsto v_i}\ (0 \leq i < n)$$

$$\frac{\epsilon; \epsilon \vdash P : tr\,(\texttt{lt}\, M\, N)}{\texttt{<}\, M\, N\, \overline{m}\, \overline{n} \mapsto \texttt{tt}_{(\texttt{lt}\, M\, N), P}}\ (m < n)$$

$$\frac{}{\texttt{test}\, c\, M\, \texttt{tt}_{M_1, M_2}\, v\, v' \mapsto v\, M_2} \qquad \frac{}{\texttt{test}\, c\, M\, \texttt{ff}_{M_1, M_2}\, v\, v' \mapsto v'\, M_2}$$

Figure 9: Selected dynamic semantic rules for arrays

6.1 Semantics of Arrays

In order to give an operational semantics to our LTT signature for arrays, we must first extend the language somewhat by adding array values. These look like $[v_0, \ldots, v_{n-1}]$ where the v_i are all values having the same type. We also need special annotated boolean values to inhabit types of the form $S_o(M)$; these will look like either $\texttt{tt}_{M,P}$, where P is a proof of M, or $\texttt{ff}_{M,P}$, where P is a refutation of M. For the purposes of our operational semantics, we will also consider "partial applications" of term constants — those whose result type is still a function or Π-type — to be values if the argument terms are values.

Since our array signature did not contain any stateful operations, we may formulate our operational semantics for that signature as a reduction relation $\mapsto$ on program terms. We take as our starting point the usual call-by-value operational rules for the core constructs of LTT which, due to lack of space, we will not reproduce here. We add to these a collection of rules defining reduction of fully-applied term constants; a few of these new rules are shown in Figure 9.

A complete set of rules for reducing fully-applied term constants, together with standard rules for call-by-value beta reduction, completely defines the run-time behavior of our LTT array signature. The next step is to prove that the language is type-safe; we do this by means of the standard Subject Reduction and Progress lemmas. The proof of Subject Reduction is standard; we will show one case of Progress to illustrate the critical dependence of this lemma on the adequacy property (introduced in Section 2) of the representation of arithmetic.

Lemma 6.1 (Subject Reduction) *If* $\Gamma \vdash e : \tau$ *and* $e \mapsto e'$*, then* $\Gamma \vdash e' : \tau$.

Lemma 6.2 (Progress) *If* $\epsilon \vdash e : \tau$ *then either* e *is a value or there exists an* e' *such that* $e \mapsto e'$.

Proof (one case): Suppose $e = \texttt{mkarray}\,\tau\, M_{sz}\, M_{pf}\, v_{sz}\, v$. Using inversion, we can establish that $M_{sz} : \mathit{Int}$, $M_{pf} : tr(\texttt{not}(\texttt{lt}\, M_{sz}\, \texttt{z}))$, $v_{sz} : S_{Int}\, M_{sz}$ and $v : \tau$. By adequacy [8], the canonical form M'_{sz} of M_{sz} must be the representation of some arithmetic expression E. Again by adequacy, the canonical form M'_{pf} of M_{pf} must be the representation of a valid derivation of $\vdash \neg(E < 0)$. By the soundness of arithmetic, it must be the case that $[\![E]\!] \geq 0$. Furthermore, we

can prove a canonical forms lemma that states that since $v_{sz} : S_{Int}\, M_{sz}$, $v_{sz} = \overline{[\![E]\!]}$. Thus $v_{sz} = \overline{n}$ and $n \geq 0$, so the evaluation rule given in Figure 9 applies.

6.2 Semantics of References

Unlike the array signature just discussed, our memory management signature contains operations whose behavior is intended to be stateful. Rather than attempt to formulate the operational semantics in terms of a reduction relation on terms, therefore, we will account for state by giving a transition relation on *machine configurations* (H, e) where H is a heap—a mapping from memory locations to values—and e is a term. We will extend the signature such that for each closed type τ that is well-formed under the signature's termless portion, there is a set of *locations* $\ell_{\tau,i}$ and corresponding object constants $\mathtt{loc}_{\ell_{\tau,i}} : \mathit{cell}$ such that $\ell_{\tau,i} : \mathit{ref}\,\tau\,\mathtt{loc}_{\ell_{\tau,i}}$. For each location ℓ we will also assume the existence of a linear constant $\mathtt{good}_\ell$, but not all of these will always be present in the linear signature. Rather, we will ensure that the linear signature in which a configuration is typed will always contain `good` constants for exactly those memory locations that are currently live. To this end we define a well-formedness judgement for heaps with the rule

$$\frac{\begin{array}{l} R = \{\mathtt{good}_{\ell_1} : \mathit{valid}\,\mathtt{loc}_{\ell_1}, \ldots, \mathtt{good}_{\ell_m} : \mathit{valid}\,\mathtt{loc}_{\ell_m}\} \\ \epsilon; \epsilon \vdash_\emptyset v_j : \tau_j \quad (\text{where } \ell_j = \ell_{\tau_j, i_j},\ 1 \leq j \leq m) \end{array}}{\vdash_R \{\ell_1 \mapsto v_1, \ldots, \ell_m \mapsto v_m\}}$$

Now we can write the typing rule for machine configurations as follows:

$$\frac{\vdash_R H \quad \epsilon; \epsilon \vdash_R e : \tau}{\vdash (H, e) : \tau}$$

The transition relation $\mapsto$ on configurations consists, as before, of the expected call-by-value rules augmented with rules for the special constructs of the signature. One such new rule is

$$\frac{H = H' \uplus \{\ell \mapsto v\}}{(H, \mathtt{free}\, A\, c\, \tau\, \tau'\, \ell\, k \hat{}\langle\!\langle M, M' \rangle\!\rangle) \mapsto (H', k \hat{}\, M')}$$

Note that the premise prevents the evaluation of an application of `free` if the named location is not currently live in the heap—in other words, attempts to free invalid locations get stuck. Thus in order to prove the Progress part of type safety we must be able to show, among other things, that this premise will be satisfied for any well-typed application of `free`.

Note that, according to the typing rule for machine configurations, the set of live locations in the heap determines the linear signature R in which the term e must be well-typed. Since all the resources in R must be consumed by the computation e, this typing rule may not apply to all the subterms of e if more than one involves stateful computation. Nevertheless, we can prove both Subject Reduction and Progress by generalizing the induction hypothesis. As before, we will show one case of Progress to illustrate the role of the linear signature.

Lemma 6.3 (Subject Reduction) *If $H = H_1 \uplus H_2$ and $\vdash_{R_1} H_1$ and $\vdash_{R_2} H_2$ and $\Gamma; \epsilon \vdash_{R_1} e : \tau$ and $(H, e) \mapsto (H', e')$ then there exist R_1' and H_1' such that $H' = H_1' \uplus H_2$ and $\vdash_{R_1'} H_1'$ and $\Gamma; \epsilon \vdash_{R_1'} e' : \tau$.*

Lemma 6.4 (Progress) *If $H = H_1 \uplus H_2$ and $\vdash_{R_1} H_1$ and $\vdash_{R_2} H_2$ and $\epsilon; \epsilon \vdash_{R_1} e : \tau$ then either e is a value or there exist H' and e' such that $(H, e) \mapsto (H', e')$.*

Proof (one case): Suppose $e = (\mathtt{free}\, A\, c\, \tau\, \tau'\, v\, k) \hat{}\, M$. We can determine by inversion that $c : \mathit{cell}$, $v : \mathit{ref}\,\tau\, c$ and $M : (\mathit{valid}\, c \otimes A)$. Canonical forms lemmas tell us that v is some label ℓ, that $c = \mathtt{loc}_\ell$ and that M is equivalent to $\langle\!\langle M_1, M_2 \rangle\!\rangle$ where $M_1 : \mathit{valid}\, c$. But the only canonical form of family $\mathit{valid}\,\mathtt{loc}_\ell$ is the constant $\mathtt{good}_\ell$; since the term is well-typed, the linear signature R_1 must include $\mathtt{good}_\ell$, so by inspection of the typing rule for heaps we conclude that a binding of the form $\ell \mapsto v'$ must appear in the heap H_1, and therefore in H, so the above rule applies.

7 Related Work and Conclusion

Concurrently with our work, Shao *et al.* [21] proposed a framework for certified code very similar in spirit to ours, which we refer to here as SSTP. While LTT is constructed by attaching LF to a typed programming language, SSTP attaches the Calculus of Inductive Constructions [18] instead. LF and Inductive Constructions are similar in that both have been widely used to formalize mathematics and both enjoy mature, robust implementations. However, there are significant differences between the systems that result in important differences in usage and expressive power between LTT and SSTP.

Regarding usage, LF is designed to encode logics, and emphasizes the notion of an adequacy theorem (Recall Section 6) stating that a canonically formed proof term of appropriate type *is* (modulo encoding) a proof in the logic being encoded. Inductive Constructions, on the other hand, does not support the same notion of canonical forms as LF, and so the LF style of encoding does not work; instead, one implements the semantics of a logic.

The differences in usage between LTT and SSTP essentially boil down to these methodological differences between LF and Constructions. In both cases, a logic and its encoding must be sound in order for the resulting language to be type-safe. In particular, when a certain condition needs to hold in order for evaluation to proceed (such as, for example, the constraints on the subscript that must be satisfied for an array operation to make sense), it must be shown as part of the safety proof that the existence of the "proof" object supplied by the program entails the required condition. In SSTP, the safety proof must characterize all the possible typing derivations of all the possible terms that could be provided, and demonstrate that the existence of any of these implies the desired property. The first of these two steps corresponds roughly to the adequacy property of an LF representation, and the second corresponds to soundness of the logic being encoded. As we have mentioned, adequacy proofs for LF are usually simple, and in some cases (such as that of arithmetic in the array example), the soundness of the logic has already been established and need not be proven again.

Regarding expressiveness, LTT is based on Linear LF, which gives LTT the power to reason about ephemeral properties, an ability SSTP does not have. We conjecture that an extension of SSTP with linear typing constructs would have similar expressive power, but the metatheory of the Calculus of Inductive Constructions with linearity has not been investigated and may be difficult. On the other hand, Inductive Constructions supports primitive recursion, which

LF does not (as such a construct would destroy LF's notion of canonical forms). Consequently, SSTP can use primitive recursion on encoded types to support intensional type analysis [9, 5], which LTT cannot. Various proposals have been made for extending LF with primitive recursion [6, 20, 19] and we are exploring integrating one of these into LTT.

LTT provides the power for very expressive type systems by allowing operations to demand proofs of arbitrary propositions (thereby escaping the usual restrictions of decidable typechecking), and by allowing operations to demand linear resources, accessible only in certain states. In this paper we have given some examples of how these facilities can be used separately; by combining them one can obtain even greater expressive power. For example, the Capability Calculus of Walker *et al.* [26] is similar to the memory management example in Section 5 in that it provides revocable access to memory cells, but it goes further with an algebra over capabilities and an ability to declare some constraints between capabilities. Those facilities can easily be encoded in LTT by an appropriate choice of judgements (representing equivalences and constraints) and proof terms. One such encoding is given in Appendix A. Beyond this, we conjecture that the security automata type system of Walker [25] and the alias-tracking type system of Smith *et al.* [22] can easily be encoded in LTT as well. Although these systems exist independently, LTT provides a uniform framework in which they can all be cast, and has the additional benefit that type-checking is decidable for any LTT encoding even if (as is the case for the Capability Calculus) decidability for the pre-existing system has not been established.

The use of LTT as a framework for certified code type theories promotes a sort of re-use, namely that of the typechecking software. A similar effort to LTT in this respect is the TinkerType meta-language of Levin and Pierce [12]. Unlike LTT, which casts everything in a common language, TinkerType emphasizes modular development of comparatively dissimilar type systems (such as F, $F_{\leq}$, F_{ω}, *etc.*). Like LTT, TinkerType provides for modular development of typechecking software; however, its main feature is support for inheritance; for example $F_{\leq}$ inherits from F and adds subtyping. LTT makes no effort at supporting inheritance; each LTT signature, with its operational semantics, must be fully developed on its own. Adding this is an interesting avenue for future work.

Acknowledgements

We would like to thank Robert Harper, Frank Pfenning and Zhong Shao for many useful conversations and suggestions.

References

[1] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, Jan. 2000.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Dec. 1995.

[3] I. Cervesato and F. Pfenning. A linear logical framework. In *Eleventh IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996.

[4] C. Colby, P. Lee, G. Necula, and F. Blau. A certifying compiler for Java. In *2000 SIGPLAN Conference on Programming Language Design and Implementation*, pages 95–107, Vancouver, British Columbia, June 2000.

[5] K. Crary and S. Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, pages 233–248, Paris, Sept. 1999.

[6] J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163, Nancy, France, Apr. 1997. Springer-Verlag. Extended version published as CMU technical report CMU-CS-96-172.

[7] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[8] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, Jan. 1993.

[9] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, Jan. 1995.

[10] R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Carnegie Mellon University, School of Computer Science, July 2000.

[11] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[12] M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. Technical Report MS-CIS-99-19, Dept of CIS, University of Pennsylvania, July 1999.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[14] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[15] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, Jan. 2002. An earlier version appeared in the 1998 Workshop on Types in Compilation, volume 1473 of Lecture Notes in Computer Science.

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[17] G. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

[18] C. Paulin-Mohring. Inductive definitions in the system coq—rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[19] C. Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, Oct. 2000.

[20] C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Fifteenth International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, Lindau, Germany, July 1998. Springer-Verlag.

$$
\begin{array}{ll}
rgn : P & \\
\mathrm{nu}_1, \mathrm{nu}_2, \ldots : rgn & r ::= \nu \\
 & \\
cap : P & \\
\mathrm{null} : cap & C ::= \emptyset \\
\mathrm{one} : rgn \to cap & \quad | \ \{r^1\} \\
\mathrm{plus} : rgn \to cap & \quad | \ \{r^+\} \\
\mathrm{join} : cap \to cap \to cap & \quad | \ C \oplus C' \\
\mathrm{bar} : cap \to cap & \quad | \ \overline{C}
\end{array}
$$

Figure 10: Formulas of the capability calculus

[21] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, pages 217–232, Portland, Oregon, Jan. 2002.

[22] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, Berlin, Germany, Mar. 2000.

[23] J. C. Vanderwaart and K. Crary. A simplified account of the metatheory of linear LF. Technical Report CMU-CS-01-154, Carnegie Mellon University, School of Computer Science, 2002.

[24] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[25] D. Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, Boston, Jan. 2000.

[26] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4), July 2000. An earlier version appeared in the 1999 Symposium on Principles of Programming Languages.

[27] H. Xi and R. Harper. A dependently typed assembly language. In *2001 ACM International Conference on Functional Programming*, pages 169–180, Florence, Italy, Sept. 2001.

[28] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

A Representing the Capability Calculus

In this appendix we present a representation of the Capability Calculus [26] in LTT. For reasons of space, we cannot present the Capability Calculus itself here; instead, we will assume the reader is familiar with it already and describe only the LTT representation. Readers not well acquainted with the Capability Calculus may skip this appendix.

One aspect of the Capability Calculus we do not address is the storage of functions in the heap. Recall that function types in the Capability Calculus are annotated with the region in which the function resides; since the functions are allowed to be polymorphic, this is difficult to represent in LTT and so we will use LTT's built-in function types instead, losing the region information. This would not be a problem in practice, as we would probably want to use LTT to represent programs after closure conversion, when functions are represented explicitly as tuples in the dynamic heap and the polymorphic code is relegated to a statically allocated text segment.

$$
\begin{array}{rl}
\text{eqcongnull}: & eqcap\ \mathrm{null}\ \mathrm{null} \\
\text{eqcongjoin}: & \Pi c_1, c_2, c_1', c_2' : cap.\, eqcap\ c_1\ c_1' \to eqcap\ c_2\ c_2' \to \\
 & \quad eqcap\ (\mathrm{join}\ c_1\ c_2)\ (\mathrm{join}\ c_1'\ c_2') \\
\text{eqcongbar}: & \Pi c, c' : cap.\, eqcap\ c\ c' \to eqcap\ (\mathrm{bar}\ c)\ (\mathrm{bar}\ c') \\
\text{eqrefl}: & \Pi c : cap.\, eqcap\ c\ c \\
\text{eqsymm}: & \Pi c, c' : cap.\, eqcap\ c\ c' \to eqcap\ c'\ c \\
\text{eqtrans}: & \Pi c_1, c_2, c_3 : cap.\, eqcap\ c_1\ c_2 \to eqcap\ c_2\ c_3 \to \\
 & \quad eqcap\ c_1\ c_3 \\
\text{eqident}: & \Pi c : cap.\, eqcap\ (\mathrm{join}\ \mathrm{null}\ c)\ c \\
\text{eqcommut}: & \Pi c, c' : cap.\, eqcap\ (\mathrm{join}\ c\ c')\ (\mathrm{join}\ c'\ c) \\
\text{eqjoinidem}: & \Pi c : cap.\, eqcap (\mathrm{bar}\ c)\ (\mathrm{join}\ (\mathrm{bar}\ c)\ (\mathrm{bar}\ c)) \\
\text{eqbarnull}: & eqcap\ (\mathrm{bar}\ \mathrm{null})\ \mathrm{null} \\
\text{eqbarone}: & \Pi r : rgn.\, eqcap\ (\mathrm{bar}(\mathrm{one}\ r))\ (\mathrm{plus}\ r) \\
\text{eqbarplus}: & \Pi r : rgn.\, eqcap\ (\mathrm{bar}(\mathrm{plus}\ r))\ (\mathrm{plus}\ r) \\
\text{eqbarbar}: & \Pi c : cap.\, eqcap\ (\mathrm{bar}\ (\mathrm{bar}\ c))\ (\mathrm{bar}\ c) \\
\text{eqbarjoin}: & \Pi c, c' : cap. \\
 & \quad eqcap\ (\mathrm{bar}\ (\mathrm{join}\ c\ c'))\ (\mathrm{join}\ (\mathrm{bar}\ c)\ (\mathrm{bar}\ c')) \\
 & \\
\text{subeq}: & \Pi c, c' : cap.\, eqcap\ c\ c' \to subcap\ c\ c' \\
\text{subtrans}: & \Pi c_1, c_2, c_3 : cap.\, subcap\ c_1\ c_2 \to subcap\ c_2\ c_3 \to \\
 & \quad subcap\ c_1\ c_3 \\
\text{subdepth}: & \Pi c_1, c_2, c_1', c_2' : cap.\, subcap\ c_1\ c_1' \to subcap\ c_2\ c_2' \to \\
 & \quad subcap\ (\mathrm{join}\ c_1\ c_2)\ (\mathrm{join}\ c_1'\ c_2') \\
\text{subcongbar}: & \Pi c, c' : cap.\, subcap\ c\ c' \to subcap\ (\mathrm{bar}\ c)\ (\mathrm{bar}\ c') \\
\text{subweakbar}: & \Pi c : cap.\, subcap\ c\ (\mathrm{bar}\ c)
\end{array}
$$

Figure 11: Proof constructors for the capability calculus.

A.1 Formulas

To begin, we must represent the syntax of capabilities and regions in the proof language of LTT. For regions this is easy, since all we need is a family *rgn* and an infinite family of region literals to represent the region names (ν) found in the Capability Calculus. The syntax of capabilities is a bit more complex, but not difficult to represent. The constants encoding the syntax of capabilities and regions are shown in Figure 10.

A.2 Judgements

The static semantics of the Capability Calculus explicitly defines two special judgements for capabilities: equality and the subcapability relation. Following the judgements-as-types paradigm, we introduce the family constants *eqcap* and *subcap* into our signature with the following kinds:

$$
\begin{array}{l}
eqcap : cap \to cap \to P \\
subcap : cap \to cap \to P
\end{array}
$$

There is, however, a third judgement form that plays a critical role in the Capability Calculus, namely the judgement that a particular capability C is the one currently held by the program. There are no inference rules in the calculus for deriving proofs of this judgement form, but the capability that appears to the left of the turnstile in the term typing rules corresponds to an *assumption* of such a judgement, much as the variable bindings in the typing context represent assumptions that typing judgements hold for certain values. We will represent this judgement using the family constant

$$hold : cap \to P.$$

Since some of the memory-management operations of the calculus replace the currently held capability with a different one, assumptions of *hold* judgements will appear in the linear rather than the intuitionistic context.

$$
\begin{array}{rl}
\texttt{0}, \texttt{1}, \ldots: & \mathit{int} \\
+, -, \times: & \mathit{int} \to \mathit{int} \to \mathit{int} \\
\texttt{if0}: & \Pi c{:}\mathit{cap}.\ \mathit{int} \to (\mathit{hold}\, c \multimap \mathit{answer}) \to (\mathit{hold}\, c \multimap \mathit{answer}) \to \mathit{hold}\, c \multimap \mathit{answer} \\
\texttt{halt}: & \Pi c{:}\mathit{cap}.\ \mathit{eqcap}\, c\, \mathrm{null} \to \mathit{int} \to \mathit{hold}\, c \multimap \mathit{answer} \\
 & \\
\texttt{newrgn}: & \Pi c{:}\mathit{cap}.(\forall r{:}\mathit{rgn}.\mathit{hndl}\, r \to \mathit{hold}\,(\mathrm{join}\, c\,(\mathrm{one}\, r)) \multimap \mathit{answer}) \to (\mathit{hold}\, c) \multimap \mathit{answer} \\
\texttt{alloc}_i: & \Pi c, c'{:}\mathit{cap}.\ \Pi r{:}\mathit{rgn}.\forall\alpha_1{:}T.\ldots\forall\alpha_i{:}T.\mathit{subcap}\, c\,(\mathrm{join}\, c'\,(\mathrm{one}\, r)) \to \\
 & \quad \mathit{hndl}\, r \to \alpha_1 \to \cdots \alpha_i \to (\mathit{tup}_i\, \alpha_1 \cdots \alpha_i\, r \to \mathit{hold}\, c \multimap \mathit{answer}) \to \mathit{hold}\, c \multimap \mathit{answer} \\
\texttt{pi}_{i,j}: & \Pi c, c'{:}\mathit{cap}.\ \Pi r{:}\mathit{rgn}.\forall\alpha_1{:}T.\ldots\forall\alpha_i{:}T.\mathit{subcap}\, c\,(\mathrm{join}\, c'\,(\mathrm{one}\, r)) \to \\
 & \quad \mathit{tup}_i\, \alpha_1 \cdots \alpha_i\, r \to (\alpha_j \to \mathit{hold}\, c \multimap \mathit{answer}) \to \mathit{hold}\, c \multimap \mathit{answer} \\
\texttt{freergn}: & \Pi c, c'{:}\mathit{cap}.\ \Pi r{:}\mathit{rgn}.\ \mathit{eqcap}\, c\,(\mathrm{join}\, c'\,(\mathrm{one}\, r)) \to \mathit{hndl}\, r \to (\mathit{hold}\, c' \multimap \mathit{answer}) \to \mathit{hold}\, c \multimap \mathit{answer}
\end{array}
$$

Figure 12: Term constants for the Capability Calculus.

A.3 Proof Constructors

The proof constructors for the capability calculus representation are shown in Figure 11. They are in direct correspondence with the constructor equality and subcapability rules of the capability calculus itself.

A.4 Type Constructors

The constructor part of the signature must provide for all the forms of types in the Capability Calculus that do not correspond directly to the "built-in" types of LTT. In particular, it must provide a type *answer* to be the return type for functions in continuation-passing style, a type of integers, types of region handles (which are used when allocating in a region and when freeing a region), and types for pointers to tuples in specific regions of the heap. Since the Capability Calculus allows tuples to be of any size, we need an infinite family of constructor constants to describe their types. Thus the constructor constants for our capability calculus representation are as follows:

$$
\begin{array}{l}
\mathit{answer} : T \\
\mathit{int} : T \\
\mathit{hndl} : \mathit{rgn} \to T \\
\mathit{tup}_i : \underbrace{T \to \cdots T \to}_{i} \mathit{rgn} \to T
\end{array}
$$

A.5 Terms

The term constants for the Capability Calculus representation are shown in Figure 12. They include all of the special term-level constructs of the Capability Calculus. The first group in the figure are the constructs that do not deal directly with memory management: integer literals, arithmetic operations, the `if0-then-else` conditional construct and the primitive `halt` which appears in the top-level continuation of a program.
The second group contains the memory-management operations of the Capability Calculus: region allocation, tuple allocation, tuple element projection and region deallocation. Note that since we have an infinite family of *tup* constructors with which to form tuple types, we also need an infinite family of `alloc` operations (indexed by the size of the tuple being allocated) and an infinite family of `pi` operations (indexed by both the tuple size and the position being read). The types for all of these operations are determined by examination of the typing rules for the corresponding expression constructs in the capability calculus; notice the use of *hold* on the left of each $\multimap$ to represent the capability that appears on the left of each turnstile in the typing rules of the Capability Calculus.

B Typing Rules for LTT

Convention In rules having an antecedent of the form $(\Gamma, X_1{:}E_1, \ldots, X_m{:}E_m); (\Delta, X_{m+1}{:}E_{m+1}, \ldots, X_n{:}E_n) \vdash \mathcal{J}$, there is an implicit side-condition that each X_i is distinct and not contained in $\mathit{Dom}(\Gamma)$ or $\mathit{Dom}(\Delta)$. A similar side-condition applies in rules with only an intuitionistic context.

$\boxed{\vdash_S \Gamma\ \mathsf{context}}$

$$
\frac{}{\vdash_S \epsilon\ \mathsf{context}} \qquad \frac{\vdash_S \Gamma\ \mathsf{context} \quad \Gamma \vdash_S k\ \mathsf{kind}}{\vdash_S \Gamma, \alpha{:}k\ \mathsf{context}}\ (\alpha \notin \mathit{Dom}(\Gamma)) \qquad \frac{\vdash_S \Gamma\ \mathsf{context} \quad \Gamma \vdash_S \tau : T}{\vdash_S \Gamma, x{:}\tau\ \mathsf{context}}\ (x \notin \mathit{Dom}(\Gamma))
$$

$$
\frac{\vdash_S \Gamma\ \mathsf{context} \quad \Gamma \vdash_S K\ \mathsf{pkind}}{\vdash_S \Gamma, a{:}K\ \mathsf{context}}\ (a \notin \mathit{Dom}(\Gamma)) \qquad \frac{\vdash_S \Gamma\ \mathsf{context} \quad \Gamma \vdash_S A : P}{\vdash_S \Gamma, u{:}A\ \mathsf{context}}\ (u \notin \mathit{Dom}(\Gamma))
$$

$\boxed{\Gamma \vdash_S k\ \mathsf{kind}}$

$$
\frac{}{\Gamma \vdash_S T\ \mathsf{kind}} \qquad \frac{\Gamma \vdash_S k_1\ \mathsf{kind} \quad \Gamma \vdash_S k_2\ \mathsf{kind}}{\Gamma \vdash_S k_1 \to k_2\ \mathsf{kind}} \qquad \frac{\Gamma \vdash_S K\ \mathsf{pkind} \quad \Gamma, a{:}K \vdash_S k\ \mathsf{kind}}{\Gamma \vdash_S \Pi a{:}K.k\ \mathsf{kind}} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S k\ \mathsf{kind}}{\Gamma \vdash_S \Pi u{:}A.k\ \mathsf{kind}}
$$

$\boxed{\Gamma \vdash_S k_1 = k_2\ \mathsf{kind}}$

$$
\frac{\Gamma \vdash_S k_1 = k_2\ \mathsf{kind} \quad \Gamma \vdash_S k_1' = k_2'\ \mathsf{kind}}{\Gamma \vdash_S k_1 \to k_1' = k_2 \to k_2'\ \mathsf{kind}} \qquad \frac{\Gamma \vdash_S K_1 = K_2\ \mathsf{pkind} \quad \Gamma \vdash_S k_1 = k_2\ \mathsf{kind}}{\Gamma \vdash_S \Pi a{:}K_1.k_1 = \Pi a{:}K_2.k_2\ \mathsf{kind}} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma, u{:}A_1 \vdash_S k_1 = k_2\ \mathsf{kind}}{\Gamma \vdash_S \Pi u{:}A_1.k_1 = \Pi u{:}A_2.k_2\ \mathsf{kind}}
$$

$$\frac{}{\Gamma \vdash_S T = T \text{ kind}} \qquad \frac{\Gamma \vdash_S k \text{ kind}}{\Gamma \vdash_S k = k \text{ kind}} \qquad \frac{\Gamma \vdash_S k_2 = k_1 \text{ kind}}{\Gamma \vdash_S k_1 = k_2 \text{ kind}} \qquad \frac{\Gamma \vdash_S k_1 = k_2 \text{ kind} \quad \Gamma \vdash_S k_2 = k_3 \text{ kind}}{\Gamma \vdash_S k_1 = k_3 \text{ kind}}$$

$\boxed{\Gamma \vdash_S c : k}$

$$\frac{}{\Gamma \vdash_S \alpha : k} \ (\Gamma(\alpha) = k) \qquad \frac{}{\Gamma \vdash_S ck : k} \ (S(ck) = k) \qquad \frac{\Gamma \vdash_S k_1 \text{ kind} \quad \Gamma, \alpha{:}k_1 \vdash_S c : k_2}{\Gamma \vdash_S \lambda\alpha{:}k_1.c : k_1 \to k_2} \qquad \frac{\Gamma \vdash_S c_1 : k_1 \to k_2 \quad \Gamma \vdash_S c_2 : k_1}{\Gamma \vdash_S c_1 c_2 : k_2}$$

$$\frac{\Gamma \vdash_S K \text{ pkind} \quad \Gamma, a{:}K \vdash_S c : k}{\Gamma \vdash_S \lambda a{:}K.c : \Pi a{:}K.k} \qquad \frac{\Gamma \vdash_S c : \Pi a{:}K.k \quad \Gamma \vdash_S A : K}{\Gamma \vdash_S c\,A : k[A/a]} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S c : k}{\Gamma \vdash_S \lambda u{:}A.c : \Pi u{:}A.k}$$

$$\frac{\Gamma \vdash_S c : \Pi u{:}A.k \quad \Gamma; \epsilon \vdash_{S,\emptyset} M : A}{\Gamma \vdash_S c\,M : k[M/u]} \qquad \frac{\Gamma \vdash_S \tau_1 : T \quad \Gamma \vdash_S \tau_2 : T}{\Gamma \vdash_S \tau_1 \to \tau_2 : T} \qquad \frac{\Gamma \vdash_S \tau_1 : T \quad \Gamma \vdash_S \tau_2 : T}{\Gamma \vdash_S \tau_1 \times \tau_2 : T} \qquad \frac{\Gamma \vdash_S k \text{ kind} \quad \Gamma, \alpha{:}k \vdash \tau : T}{\Gamma \vdash_S \Pi\alpha{:}k.\tau : T}$$

$$\frac{\Gamma \vdash_S k \text{ kind} \quad \Gamma, \alpha{:}k \vdash \tau : T}{\Gamma \vdash_S \Sigma\alpha{:}k.\tau : T} \qquad \frac{\Gamma \vdash_S K \text{ pkind} \quad \Gamma, a{:}K \vdash \tau : T}{\Gamma \vdash_S \Pi a{:}K.\tau : T} \qquad \frac{\Gamma \vdash_S K \text{ pkind} \quad \Gamma, a{:}K \vdash \tau : T}{\Gamma \vdash_S \Sigma a{:}K.\tau : T} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash \tau : T}{\Gamma \vdash_S \Pi u{:}A.\tau : T}$$

$$\frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash \tau : T}{\Gamma \vdash_S \Sigma u{:}A.\tau : T} \qquad \frac{\Gamma \vdash_S A : P^+ \quad \Gamma \vdash_S \tau : T}{\Gamma \vdash_S A \multimap \tau : T} \qquad \frac{\Gamma \vdash_S c : k \quad \Gamma \vdash_S k = k' \text{ kind}}{\Gamma \vdash_S c : k'}$$

$\boxed{\Gamma \vdash_S c_1 = c_2 : k}$

$$\frac{}{\Gamma \vdash_S \alpha = \alpha : k} \ (\Gamma(\alpha) = k) \qquad \frac{}{\Gamma \vdash_S ck = ck : k} \ (S(ck) = k) \qquad \frac{\Gamma, \alpha{:}k_1 \vdash_S c_1 = c_2 : k_2 \quad \Gamma \vdash_S k_1' = k_1 \text{ kind} \quad \Gamma \vdash_S k_1'' = k_1 \text{ kind}}{\Gamma \vdash_S \lambda\alpha{:}k_1'.c_1 = \lambda\alpha{:}k_1''.c_2 : k_1 \to k_2}$$

$$\frac{\Gamma \vdash_S c_1 = c_2 : k_1 \to k_2 \quad \Gamma \vdash_S c_1' = c_2' : k_1}{\Gamma \vdash_S c_1\,c_1' = c_2\,c_2' : k_2} \qquad \frac{\Gamma, a{:}K \vdash_S c_1 = c_2 : k \quad \Gamma \vdash_S K_1 = K \text{ pkind} \quad \Gamma \vdash_S K_2' = K \text{ pkind}}{\Gamma \vdash_S \lambda\alpha{:}K_1.c_1 = \lambda\alpha{:}K_2.c_2 : \Pi a{:}K.k} \qquad \frac{\Gamma \vdash_S c_1 = c_2 : \Pi a{:}K.k \quad \Gamma \vdash_S A_1 = A_2 : K}{\Gamma \vdash_S c_1\,A_1 = c_2\,A_2 : k[A_1/a]}$$

$$\frac{\Gamma, u{:}A \vdash_S c_1 = c_2 : k}{\Gamma \vdash_S \lambda u{:}A_1.c_1 = \lambda u{:}A_2.c_2 : \Pi u{:}A.k} \qquad \frac{\Gamma \vdash_S A_1 = A : P \quad \Gamma \vdash_S A_2 = A : P}{\Gamma \vdash_S c_1 = c_2 : \Pi u{:}A.k \quad \Gamma \vdash_S M_1 = M_2 : A} \qquad \frac{\Gamma \vdash_S c_1\,M_1 = c_2\,M_2 : k[M_1/u]}{\Gamma \vdash_S \tau_1 = \tau_2 : T \quad \Gamma \vdash_S \tau_1' = \tau_2' : T} \qquad \frac{\Gamma \vdash_S \tau_1 \to \tau_1' = \tau_2 \to \tau_2' : T}{\Gamma \vdash_S \tau_1 \to \tau_1' = \tau_2 \to \tau_2' : T}$$

$$\frac{\Gamma \vdash_S \tau_1 = \tau_2 : T \quad \Gamma \vdash_S \tau_1' = \tau_2' : T}{\Gamma \vdash_S \tau_1 \times \tau_1' = \tau_2 \times \tau_2' : T} \qquad \frac{\Gamma \vdash_S k_1 = k_2 \text{ kind} \quad \Gamma, \alpha{:}k_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Pi\alpha{:}k_1.\tau_1 = \Pi\alpha{:}k_2.\tau_2 : T} \qquad \frac{\Gamma \vdash_S k_1 = k_2 \text{ kind} \quad \Gamma, \alpha{:}k_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Sigma\alpha{:}k_1.\tau_1 = \Sigma\alpha{:}k_2.\tau_2 : T}$$

$$\frac{\Gamma \vdash_S K_1 = K_2 \text{ pkind} \quad \Gamma, a{:}K_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Pi a{:}K_1.\tau_1 = \Pi a{:}K_2.\tau_2 : T} \qquad \frac{\Gamma \vdash_S K_1 = K_2 \text{ pkind} \quad \Gamma, a{:}K_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Sigma a{:}K_1.\tau_1 = \Sigma a{:}K_2.\tau_2 : T} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma, u{:}A_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Pi u{:}A_1.\tau_1 = \Pi u{:}A_2.\tau_2 : T}$$

$$\frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma, u{:}A_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Sigma u{:}A_1.\tau_1 = \Sigma u{:}A_2.\tau_2 : T} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P^+ \quad \Gamma \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S A_1 \multimap \tau_1 = A_2 \multimap \tau_2 : T} \qquad \frac{\Gamma \vdash_S c_1 = c_2 : k \quad \Gamma \vdash_S k = k' \text{ kind}}{\Gamma \vdash_S c_1 = c_2 : k'}$$

$$\frac{\Gamma, \alpha{:}k_1 \vdash_S c_1 = k_2 \quad \Gamma \vdash_S c_2 : k_1}{\Gamma \vdash_S (\lambda\alpha{:}k_1.c_1)\,c_2 = c_1[c_2/\alpha] : k_2} \qquad \frac{\Gamma, a{:}K \vdash_S c : k \quad \Gamma \vdash_S A : K}{\Gamma \vdash_S (\lambda a{:}K.c)\,A = c[A/a] : k[A/a]} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S c : k \quad \Gamma \vdash_S M : A}{\Gamma \vdash_S (\lambda u{:}A.c)\,M = c[M/u] : k[M/u]}$$

$$\frac{\Gamma \vdash_S k_1 \text{ kind} \quad \Gamma, \alpha{:}k_1 \vdash_S c_1\,\alpha = c_2\,\alpha : k_2}{\Gamma \vdash_S c_1 = c_2 : k_1 \to k_2} \qquad \frac{\Gamma \vdash_S K \text{ pkind} \quad \Gamma, a{:}K \vdash_S c_1\,a = c_2\,a : k}{\Gamma \vdash_S c_1 = c_2 : \Pi a{:}K.k} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S c_1\,u = c_2\,u : k}{\Gamma \vdash_S c_1 = c_2 : \Pi u{:}A.k}$$

$$\frac{\Gamma \vdash_S c : k}{\Gamma \vdash_S c = c : k} \qquad \frac{\Gamma \vdash_S c_2 = c_2 : k}{\Gamma \vdash_S c_1 = c_2 : k} \qquad \frac{\Gamma \vdash_S c_1 = c_2 : k \quad \Gamma \vdash_S c_2 = c_3 : k}{\Gamma \vdash_S c_1 = c_3 : k}$$

$\boxed{\Gamma; \Delta \vdash_{S,R} e : \tau}$

$$\frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} x : \tau} \ (\Gamma(x) = \tau) \qquad \frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} ek : \tau} \ (S(ek) = \tau) \qquad \frac{\Gamma \vdash_S \tau_1 : T \quad \Gamma, x{:}\tau_1; \epsilon \vdash_{S,\emptyset} e : \tau_2}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \tau_1 \to \tau_2 \quad \Gamma; \Delta_2 \vdash_{S,R_2} e_2 : \tau_1}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} e_1 e_2 : \tau_2} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \tau_1 \quad \Gamma; \Delta_2 \vdash_{S,R_2} e_2 : \tau_2}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma; \Delta \vdash_{S,r} e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash_{S,R} \pi_i e : \tau_i} \ (i = 1, 2)$$

$$\frac{\Gamma \vdash_S k \text{ kind} \quad \Gamma, \alpha{:}k; \epsilon \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda\alpha{:}k.e : \Pi\alpha{:}k.\tau} \qquad \frac{\Gamma; \Delta \vdash_{S,R} e : \Pi\alpha{:}k.\tau \quad \Gamma \vdash_S c : k}{\Gamma; \Delta \vdash_{S,R} e\,c : \tau[c/\alpha]} \qquad \frac{\Gamma \vdash_S c : k \quad \Gamma, \alpha{:}k \vdash_S \tau : T \quad \Gamma; \Delta \vdash_{S,R} e : \tau[c/\alpha]}{\Gamma; \Delta \vdash_{S,R} \text{pack } \langle c, e \rangle \text{ as } \Sigma\alpha{:}k.\tau : \Sigma\alpha{:}k.\tau}$$

$$\frac{\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \Sigma\alpha{:}k.\tau \quad (\Gamma, \alpha{:}k, x{:}\tau); \Delta_2 \vdash_{S,R_2} e_2 : \tau'}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \text{let } \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \tau'} \qquad \frac{\Gamma \vdash_S K \text{ pkind} \quad \Gamma, a{:}K; \epsilon \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda a{:}K.e : \Pi a{:}K.\tau} \qquad \frac{\Gamma; \Delta \vdash_{S,R} e : \Pi a{:}K.\tau \quad \Gamma \vdash_S A : K}{\Gamma; \Delta \vdash_{S,R} e\,A : \tau[A/a]}$$

$$\frac{\Gamma \vdash_S A : K \quad \Gamma, a{:}K \vdash_S \tau : T \quad \Gamma; \Delta \vdash_{S,R} e : \tau[A/a]}{\Gamma; \Delta \vdash_{S,R} \text{pack } \langle A, e \rangle \text{ as } \Sigma a{:}K.\tau : \Sigma a{:}K.\tau} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \Sigma a{:}K.\tau \quad (\Gamma, a{:}K, x{:}\tau); \Delta_2 \vdash_{S,R_2} e_2 : \tau'}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \text{let } \langle a, x \rangle = e_1 \text{ in } e_2 : \tau'} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A; \epsilon \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda u{:}A.e : \Pi u{:}A.\tau}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} e : \Pi u{:}A.\tau \quad \Gamma; \epsilon \vdash_{S,\emptyset} M : A}{\Gamma; \Delta \vdash_{S,R} e\,M : \tau[M/u]} \qquad \frac{\Gamma; \epsilon \vdash_{S,\emptyset} M : A \quad \Gamma, u{:}A \vdash_S \tau : T \quad \Gamma; \Delta \vdash_{S,R} e : \tau[M/u]}{\Gamma; \Delta \vdash_{S,R} \text{pack } \langle M, e \rangle \text{ as } \Sigma u{:}A.\tau : \Sigma u{:}A.\tau} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \Sigma u{:}A.\tau \quad (\Gamma, u{:}A, x{:}\tau); \Delta_2 \vdash_{S,R_2} e_2 : \tau'}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \text{let } \langle u, x \rangle = e_1 \text{ in } e_2 : \tau'}$$

$$\frac{\Gamma \vdash_S A : P^+ \quad \Gamma; u \hat{:} A \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \hat{\lambda} u{:}A.e : A \multimap \tau} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} e : A \multimap \tau \quad \Gamma; \Delta_2 \vdash_{S,R_2} M : A}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} e \hat{} M : \tau} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} M : A_1 \otimes A_2 \quad \Gamma; (\Delta_2, u_1 \hat{:} A_1, u_2 \hat{:} A_2) \vdash_{S,R_2} e : \tau}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \text{let } \langle\!\langle u_1, u_2 \rangle\!\rangle = M \text{ in } e : \tau}$$

$$\frac{\Gamma; \Delta_1 \vdash_{S,R_1} M : 1 \quad \Gamma; \Delta_2 \vdash_{S,R_2} e : \tau}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \text{let } \star = M \text{ in } e : \tau} \qquad \frac{\Gamma; \Delta \vdash_{S,R} e : \tau \quad \Gamma \vdash_S \tau = \tau' : T}{\Gamma; \Delta \vdash_{S,R} e : \tau'}$$

$\boxed{\Gamma \vdash_S K \ \mathsf{pkind}}$

$$\frac{}{\Gamma \vdash_S P \ \mathsf{pkind}} \qquad \frac{}{\Gamma \vdash_S P^+ \ \mathsf{pkind}} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S K \ \mathsf{pkind}}{\Gamma \vdash_S \Pi u{:}A.K \ \mathsf{pkind}}$$

$\boxed{\Gamma \vdash_S K_1 = K_2 \ \mathsf{pkind}}$

$$\frac{\Gamma \vdash_S K \ \mathsf{pkind}}{\Gamma \vdash_S K = K \ \mathsf{pkind}} \qquad \frac{\Gamma \vdash_S K_2 = K_1 \ \mathsf{pkind}}{\Gamma \vdash_S K_1 = K_2 \ \mathsf{pkind}} \qquad \frac{\Gamma \vdash_S K_1 = K_2 \ \mathsf{pkind} \quad \Gamma \vdash_S K_2 = K_3 \ \mathsf{pkind}}{\Gamma \vdash_S K_1 = K_3 \ \mathsf{pkind}}$$

$$\frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma, u{:}A_1 \vdash_S K_1 = K_2 \ \mathsf{pkind}}{\Gamma \vdash_S \Pi u{:}A_1.K_1 = \Pi u{:}A_2.K_2 \ \mathsf{pkind}}$$

$\boxed{\Gamma \vdash_S A : K}$

$$\frac{}{\Gamma \vdash_S a : K}\ (\Gamma(a) = K) \qquad \frac{}{\Gamma \vdash_S Ak : K}\ (S(Ak) = K) \qquad \frac{\Gamma \vdash_S A_1 : P \quad \Gamma, u{:}A_1 \vdash_S A_2 : K}{\Gamma \vdash_S \lambda u{:}A_1.A_2 : \Pi u{:}A_1.K} \qquad \frac{\Gamma \vdash_S A : \Pi u{:}A'.K \quad \Gamma; \epsilon \vdash_{S,\emptyset} M : A'}{\Gamma \vdash_S A\, M : K[M/u]}$$

$$\frac{\Gamma \vdash_S A_1 : P \quad \Gamma, u{:}A_1 \vdash_S A_2 : P}{\Gamma \vdash_S \Pi u{:}A_1.A_2 : P} \qquad \frac{\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P}{\Gamma \vdash_S A_1 \multimap A_2 : P} \qquad \frac{\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P}{\Gamma \vdash_S A_1 \,\&\, A_2 : P} \qquad \frac{}{\Gamma \vdash_S \top : P}$$

$$\frac{\Gamma \vdash_S A_1 : P^+ \quad \Gamma \vdash_S A_2 : P^+}{\Gamma \vdash_S A_1 \otimes A_2 : P^+} \qquad \frac{}{\Gamma \vdash_S 1 : P^+} \qquad \frac{\Gamma \vdash_S A : P}{\Gamma \vdash_S A : P^+} \qquad \frac{\Gamma \vdash_S A : K \quad \Gamma \vdash_S K = K' \ \mathsf{pkind}}{\Gamma \vdash_S A : K'}$$

$\boxed{\Gamma \vdash_S A_1 = A_2 : K}$

$$\frac{}{\Gamma \vdash_S a = a : K}\ (\Gamma(a) = K) \qquad \frac{}{\Gamma \vdash_S Ak = Ak : K}\ (S(Ak) = K) \qquad \frac{\Gamma \vdash_S A_1 = A_2 : K \quad \Gamma \vdash_S A_2 = A_3 : K}{\Gamma \vdash_S A_1 = A_3 : K}$$

$$\frac{\Gamma \vdash_S A_1 = A_2 : \Pi u : B.K \quad \Gamma \vdash_S M_1 = M_2 : B}{\Gamma \vdash_S A_1\, M_1 = A_2\, M_2 : K[M_1/u]} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma, u{:}A_1 \vdash_S A_1' = A_2' : P}{\Gamma \vdash_S \Pi u{:}A_1.A_1' = \Pi u{:}A_2.A_2' : P} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma \vdash_S A_1' = A_2' : P}{\Gamma \vdash_S A_1 \multimap A_1' = A_2 \multimap A_2' : P}$$

$$\frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma \vdash_S A_1' = A_2' : P}{\Gamma \vdash_S A_1 \,\&\, A_1' = A_2 \,\&\, A_2' : P} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P^+ \quad \Gamma \vdash_S A_1' = A_2' : P^+}{\Gamma \vdash_S A_1 \otimes A_1' = A_2 \otimes A_2' : P^+} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P}{\Gamma \vdash_S A_1 = A_2 : P^+} \qquad \frac{\Gamma \vdash_S A_2 = A_1 : K}{\Gamma \vdash_S A_1 = A_2 : K}$$

$$\frac{\Gamma \vdash_S A : P \quad \Gamma \vdash_S A_1 = A : P \quad \Gamma \vdash_S A_2 = A : P}{\Gamma \vdash_S \lambda u{:}A_1.A_1' = \lambda u{:}A_2.A_2' : \Pi u{:}A.K} \qquad \frac{\Gamma, u{:}A \vdash_S A_1' = A_2' : K \quad \Gamma \vdash_S B : P}{\Gamma, u{:}B \vdash_S A_1 = A_2 : K \quad \Gamma; \epsilon \vdash_S M_1 = M_2 : B} \qquad \frac{\Gamma \vdash_S B : P}{\Gamma, u{:}B \vdash_S A_1\, u = A_2\, u : K}$$

$\boxed{\Gamma; \Delta \vdash_{S,R} M : A}$

$$\frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} u : A}\ (\Gamma(u) = A) \qquad \frac{}{\Gamma; u\hat{:}A \vdash_{S,\emptyset} u : A} \qquad \frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} Mk : A}\ (S(Mk) = A) \qquad \frac{}{\Gamma; \epsilon \vdash_{S,\{Mlk:A\}} Mlk : A}$$

$$\frac{\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P \quad \Gamma, u{:}A_1; \Delta \vdash_{S,R} M : A_2}{\Gamma; \Delta \vdash_{S,R} \lambda u{:}A_1.M : \Pi u{:}A_1.A_2} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_1 : \Pi u{:}A_1.A_2 \Gamma; \epsilon \vdash_{S,\emptyset} M_2 : A_1}{\Gamma; \Delta \vdash_{S,R} M_1 M_2 : A_2[M_2/u]} \qquad \frac{\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P \quad \Gamma; \Delta, u{:}A_1 \vdash_{S,R} M : A_2}{\Gamma; \Delta \vdash_{S,R} \hat{\lambda} u{:}A_1.M : A_1 \multimap A_2}$$

$$\frac{\Gamma; \Delta_1 \vdash_{S,R_1} M_1 : A_1 \multimap A_2 \quad \Gamma; \Delta_2 \vdash_{S,R_2} M_2 : A_1}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} M_1 \hat{} M_2 : A_2} \qquad \frac{\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P \quad \Gamma; \Delta \vdash_{S,R} M_1 : A_1 \quad \Gamma; \Delta \vdash_{S,R} M_2 : A_2}{\Gamma; \Delta \vdash_{S,R} \langle M_1, M_2 \rangle : A_1 \,\&\, A_2} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M : A_1 \,\&\, A_2}{\Gamma; \Delta \vdash_{S,R} \pi_i M : A_i}\ (i = 1, 2)$$

$$\frac{}{\Gamma; \Delta \vdash_{S,R} \langle\rangle : \top} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M : A \quad \Gamma \vdash_S A = A' : P}{\Gamma; \Delta \vdash_{S,R} M : A'} \qquad \frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} \star : 1} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} M_1 : A_1 \quad \Gamma; \Delta_2 \vdash_{S,R_2} M_2 : A_2}{\Gamma; \Delta_1, \Delta_2 \vdash_{S,R_1 \uplus R_2} \langle\!\langle M_1, M_2 \rangle\!\rangle : A_1 \otimes A_2}$$

$\boxed{\Gamma; \Delta \vdash_S M_1 = M_2 : A}$

$$\frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} u = u : A}\ (\Gamma(u) = A) \qquad \frac{}{\Gamma; u\hat{:}A \vdash_{S,\emptyset} u = u : A} \qquad \frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} Mk = Mk : A}\ (S(Mk) = A) \qquad \frac{}{\Gamma; \epsilon \vdash_{S,\{Mlk:A\}} Mlk = Mlk : A}$$

$$\frac{\Gamma, u{:}A \vdash_{S,R} M_1 = M_2 : B \quad \Gamma \vdash_S A_1 = A : P \quad \Gamma \vdash_S A_2 = A : P}{\Gamma; \Delta \vdash_{S,R} \lambda u{:}A_1.M_1 = \lambda u{:}A_2.M_2 : \Pi u{:}A.B} \qquad \frac{\Gamma; \epsilon \vdash_{S,\emptyset} N_1 = N_2 : A \quad \Gamma; \Delta \vdash_{S,R} M_1 = M_2 : \Pi u{:}A.B}{\Gamma; \Delta \vdash_{S,R} M_1\, N_1 = M_2\, N_2 : B[N_1/u]} \qquad \frac{\Gamma; \Delta, u\hat{:}A \vdash_{S,R} M_1 = M_2 : B \quad \Gamma \vdash_S A_1 = A : P \quad \Gamma \vdash_S A_2 = A : P}{\Gamma; \Delta \vdash_{S,R} \hat{\lambda} u{:}A_1.M_1 = \hat{\lambda} u{:}A_2.M_2 : A \multimap B}$$

$$\frac{\Gamma; \Delta_2 \vdash_{S,R_2} N_1 = N_2 : A \quad \Gamma; \Delta_1 \vdash_{S,R_1} M_1 = M_2 : A \multimap B}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} M_1 \hat{} N_1 = M_2 \hat{} N_2 : B} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A \quad \Gamma; \Delta \vdash_{S,R} N_1 = N_2 : B}{\Gamma; \Delta \vdash_{S,R} \langle M_1, N_1 \rangle = \langle M_2, N_2 \rangle : A \,\&\, B} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A_1 \,\&\, A_2}{\Gamma; \Delta \vdash_{S,R} \pi_i M_1 = \pi_i M_2 : A_i}\ (i = 1, 2)$$

$$\frac{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A' \quad \Gamma \vdash_S A' = A : P}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_2 = M_1 : A}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A \quad \Gamma; \Delta \vdash_{S,R} M_2 = M_3 : A}{\Gamma; \Delta \vdash_{S,R} M_1 = M_3 : A}$$

$$\frac{\Gamma, u{:}A; \Delta \vdash_{S,R} M_1 = M_2 : B \quad \Gamma \vdash_S A : P \quad \Gamma; \epsilon \vdash_{S,\emptyset} N_1 = N_2 : A}{\Gamma; \Delta \vdash_{S,R} (\lambda u{:}A.M_1)\, N_1 = M_2[N_2/u] : B[N_1/u]} \qquad \frac{\Gamma; \Delta_1, u\hat{:}A \vdash_{S,R_1} M_1 = M_2 : B \quad \Gamma \vdash_S A : P \quad \Gamma; \Delta_2 \vdash_{S,R_2} N_1 = N_2 : A}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} (\hat{\lambda} u{:}A.M_1) \hat{} N_1 = M_2[N_2/u] : B}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} M_1 = N_1 : A_1 \quad \Gamma; \Delta \vdash_{S,R} M_2 = N_2 : A_2}{\Gamma; \Delta \vdash_{S,R} \pi_i \langle M_1, M_2 \rangle = N_i : A_i} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A; \Delta \vdash_{S,R} M_1\, u = M_2\, u : B}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : \Pi u{:}A.B} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma; \Delta, u\hat{:}B \vdash_{S,R} M_1 \hat{} u = M_2 \hat{} u : B}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A \multimap B}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} \pi_1 M_1 = \pi_1 M_2 : A \quad \Gamma; \Delta \vdash_{S,R} \pi_2 M_1 = \pi_2 M_2 : B}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A \,\&\, B} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_1 : \top \quad \Gamma; \Delta \vdash_{S,R} M_2 : \top}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : \top}$$