An Expressive, Scalable Type Theory for Certified Code

Karl Crary Joseph C. Vanderwaart

May 1, 2001
CMU-CS-01-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 provide additional re-use with a framework for modular development of operational semantics. This framework allows independent type systems and their operational semantics to be joined together, automatically inheriting the type safety properties of those individual systems.

This material is based on work supported in part by ARPA grant F-19628-95-C-0050 and NSF grant CCR-9984812. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of these agencies.

Keywords: Certified code, logical frameworks, type theory, linear logic

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 [11] 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) [15, 1], and the *type-theoretic* approach exemplified by Typed Assembly Language (TAL) [14]. 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) [7] 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 [13, 5, 16, 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. In addition, although new types in PCC certainly must be proven sound, in Appel and Felty's PCC implementation [1] those proofs are done within the logic and often do not interact with each other, making it possible to develop them in a modular fashion. 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 proposition and proof (encoded in types and terms using the propositions-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 LF. 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 proof types and types for proof terms), 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.

In addition, to further improve scalability, we present a framework for modular development of operational semantics. This framework allows for fragments of a signature to be given operational semantics and shown type-safe independently, in such a way that those independent semantics can ultimately be glued together into a full semantics and safety proof. Consequently, when a new component is to be added to a type system, its safety proof can be done alone, without any need to re-do other safety proofs.

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 [21, 20]. 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 mostly 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 [12], 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 Crary, *et al.*'s region type system [5].

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 present the framework for modular development of operational semantics in Section 6.

This paper assumes familiarity with linear logic [6, 18] and with the propositions-as-types correspondence [9]. Additional familiarity with LF and logical frameworks in general will be helpful, but is not required.

2 Intuitionistic LTT

The LTT type theory consists of two parts: a proof sub-language, 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 unmodified LF. 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 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

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

Figure 1: Proof Language Syntax

$$
\begin{array}{llcl}
\textit{kinds} & k & ::= & T \mid k_1 \to k_2 \mid \Pi a{:}K.k \mid \Pi u{:}A.k \\
\textit{constructors} & c & ::= & \alpha \mid ck \mid \\
 & & & \lambda\alpha{:}k.c \mid c_1 c_2 \mid \lambda a{:}K.c \mid c\,A \mid \lambda u{:}A.c \mid c\,M \mid \\
 & & & c_1 \to c_2 \mid c_1 \times c_2 \mid \\
 & & & \Pi\alpha{:}k.c \mid \Sigma\alpha{:}k.c \mid \Pi a{:}K.c \mid \Sigma a{:}K.c \mid \Pi u{:}A.c \mid \Sigma u{:}A.c \\
\textit{terms} & e & ::= & x \mid ek \mid \\
 & & & \lambda x{:}c.e \mid e_1 e_2 \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \mid \\
 & & & \lambda\alpha{:}k.e \mid e\,c \mid \mathsf{pack}\ \langle c, e \rangle\ \mathsf{as}\ \Sigma\alpha{:}k.c \mid \mathsf{let}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 \mid \\
 & & & \lambda a{:}K.e \mid e\,A \mid \mathsf{pack}\ \langle A, e \rangle\ \mathsf{as}\ \Sigma a{:}K.c \mid \mathsf{let}\ \langle a, x \rangle = e_1\ \mathsf{in}\ e_2 \mid \\
 & & & \lambda u{:}A.e \mid e\,M \mid \mathsf{pack}\ \langle M, e \rangle\ \mathsf{as}\ \Sigma u{:}A.c \mid \mathsf{let}\ \langle u, x \rangle = e_1\ \mathsf{in}\ e_2 \\
\textit{contexts} & \Gamma & ::= & \epsilon \mid \Gamma, \alpha{:}k \mid \Gamma, x{:}c \mid \Gamma, a{:}K \mid \Gamma, u{:}A
\end{array}
$$

Figure 2: Programming Language Syntax

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 no 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 LF 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.* [7]) of a fragment of first-order logic. Suppose we wish to reason about natural numbers t given by the syntax

$$t ::= u \mid 0 \mid succ(t)$$

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:

$$
\begin{array}{lcl@{\qquad}lcl}
i & : & P & o & : & P \\
\mathtt{z} & : & i & \mathtt{eq} & : & i \to i \to o \\
\mathtt{s} & : & i \to i & \mathtt{impl} & : & o \to o \to o \\
 & & & \mathtt{all} & : & (i \to o) \to o
\end{array}
$$

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 LF; 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 $\mathtt{tr} : o \to P$. For any proposition φ implemented by M, $\mathtt{tr}\, M$ will be inhabited exactly when φ is true. It remains to give proof terms (representing inference rules of the logic) inhabiting the $\mathtt{tr}$ family. A few examples of these are:

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

For example, the judgement that $\forall u.\, u = u$ is true is implemented by the family $\mathtt{tr}\,(\mathtt{all}\,(\lambda u{:}i.\, \mathtt{eq}\, u\, u))$, 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.* [7].

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:

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.

[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. However, a set-theoretic function from B to C is defined on all members of B, as usual.

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

The judgement forms for LTT's static semantics are given in Figure 3. 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 A (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 $S \upharpoonright sk$ 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 $Ak \in Dom(S)$, $\vdash_{S \upharpoonright Ak} S(Ak)$ pkind, and
- for all $Mk \in Dom(S)$, $\vdash_{S \upharpoonright Mk} S(Mk) : P$, and
- for all $ck \in Dom(S)$, $\vdash_{S \upharpoonright ck} S(ck)$ 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.

3 Example: Integers and Array Bounds

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 verify statically that any given operation is safe, we will require that the client of our array module exhibit a proof that the index she wishes to access falls within the proper range. In order to represent such proofs in LTT, we must define a family of objects to represent integer expressions and establish a correspondence between those expressions and run-time integer values. Once we have an LF-style representation of a theory of arithmetic — any sufficiently expressive theory that may be encoded in LF will do — we use the correspondence we have set up to declare special versions of the array operations that require certain facts to have been proven about their arguments before they may be called.

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

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

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 formulae in the theory of arithmetic, 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 $N : \mathit{Int}$, then $S_{\mathit{Int}}\, N$ 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:

$$\begin{aligned} \mathtt{z} &: \mathit{Int} \\ \mathtt{s} &: \mathit{Int} \to \mathit{Int} \\ \mathtt{neg} &: \mathit{Int} \to \mathit{Int} \\ \mathtt{plus} &: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\ \mathtt{eq} &: \mathit{Int} \to \mathit{Int} \to o \\ \mathtt{lt} &: \mathit{Int} \to \mathit{Int} \to o \\ \mathtt{not} &: o \to o \\ \mathtt{and} &: o \to o \to o \\ \mathtt{and\text{-}i} &: \Pi u{:}o.\Pi v{:}o.\, tr\, u \to tr\, v \to tr\, (\mathtt{and}\, u\, v) \end{aligned}$$

As in the earlier example of our proof language, $\mathtt{z}$ represents zero, and $(\mathtt{s}\, M)$ represents the successor of the integer represented by M. The proposition $\mathtt{eq}\, M\, N$ states that the integers represented by M and N are equal, and the proposition $\mathtt{lt}\, M\, N$ states that the value of M is less than the value of N. We mention here just one proof constructor (rule of inference), `and-i`, which corresponds to the and-introduction rule of propositional logic and which we will use in an example later; 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 the array operations are not surprising:

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

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.

To begin, we add a new constant to the signature for each integer, to function as a "singleton literal." More precisely, for any integer n we will have a constant $\overline{n} : S_{\mathit{Int}}\, (\mathtt{s}^n\, \mathtt{z})$. These singleton constants may be used in place of ordinary integer constants whenever we need to prove something about a particular, statically available, number. For example, if M is a proof that five is positive then the expression $(\texttt{mkarray}\,\mathit{int}\,(\texttt{s}^5\,\texttt{z})\,M\,\overline{5}\,0)$ is well typed and produces a 5-element array of integers, initially set to all zeroes.

Next, we declare special versions of the arithmetic operations that act on values of singleton types, for example:

$$\texttt{+} : \Pi u{:}\mathit{Int}.\,\Pi v{:}\mathit{Int}.\,S_{\mathit{Int}}\,u \to S_{\mathit{Int}}\,v \to S_{\mathit{Int}}\,(\texttt{plus}\,u\,v)$$

The idea here is that if we know the values of the two operands of an addition, then we know the result.

The last things we need in order to perform simple manipulation of singleton integers are a few coercions — operations that are not intended to have any run-time significance but exist only to change the type of a value.

$$\begin{aligned} \texttt{focus} &: \mathit{int} \to \Sigma u{:}\mathit{Int}.S_{\mathit{Int}}\,u \\ \texttt{blur} &: \Pi u{:}\mathit{Int}.\,S_{\mathit{Int}}\,u \to \mathit{int} \\ \texttt{convert} &: \Pi u{:}\mathit{Int}.\,\Pi v{:}\mathit{Int}.\mathit{tr}\,(\texttt{eq}\,u\,v) \to S_{\mathit{Int}}\,u \to S_{\mathit{Int}}\,v \end{aligned}$$

The first two of these allow arbitrary integer values to be temporarily manipulated as singletons. The `focus` operator turns an integer into a singleton whose value is hidden in a dependent sum, which can then be unpacked to give an object-level name to that integer. The `blur` operator turns any singleton into an ordinary integer, effectively forgetting any information that had been proved about it. The last, `convert`, validates the intuition that if two integer expressions are known to have the same value, then elements of the corresponding singletons should be interchangeable.

Now we address the issue of incorporating information discovered by the program at run time into our proof system. Intuitively, we want a special form of conditional expression that introduces a proof of the condition into its then-branch and a refutation of the condition into its else-branch. To accomplish this, we replace ordinary booleans with special values whose types associate them with the truth or falsehood of a particular proposition, and we replace the ordinary integer comparison operators with versions that return the new kind of "boolean" values. The constructor and term constants necessary to do this are as follows:

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

For any proposition p, the sole value of type $S_o\,p$ is one that encodes the truth or falsehood of p at run time. Values of this sort of type are obtained by applying the special comparison operators `=` and `<` to singleton integers; when an application of one of these operators is evaluated, it will produce the appropriate "truth value." This value may then be examined using the `test` function, which takes a value representing the truth of some proposition and selects one of two values of some type α accordingly. The functions that are passed to `test` may use their proof arguments to certify the safety of array operations: these arguments constitute "empirical evidence" that the given statements are true or false. (An alternative design, using a disjoint union type, is to define S_o to mean $\lambda u{:}o.(\mathit{tr}\,u \times \texttt{unit}) + (\mathit{tr}\,(\texttt{not}\,u) \times \texttt{unit})$, implement `test` appropriately, and leave only `=` and `<` as primitive.)

To illustrate this mechanism, the following code fragment will return the ith element of an array provided that i is in the correct range, 0 if i is too small, and 10 if i is too large. We will assume that u is a variable of family Int, a is an array of type $\mathit{array}\,u\,\alpha$, n is the size of the array, having type $S_{\mathit{Int}}\,u$, and i has type int.

$$\begin{array}{l} \mathsf{let}\ \langle v, i' \rangle = \texttt{focus}\,i\ \mathsf{in} \\ \quad \texttt{test}\,\mathit{int}\,(\texttt{lt}\,v\,\texttt{z})\,(\texttt{<}\,v\,\texttt{z}\,i'\,\overline{0}) \\ \qquad (\lambda p{:}\mathit{tr}\,(\texttt{lt}\,v\,\texttt{z}).\,0) \\ \qquad (\lambda p{:}\mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,v\,\texttt{z})). \\ \qquad\quad \texttt{test}\,\mathit{int}\,(\texttt{lt}\,v\,u)\,(\texttt{<}\,v\,u\,i'\,n) \\ \qquad\qquad (\lambda q{:}\texttt{tr}\,(\texttt{lt}\,v\,u). \\ \qquad\qquad\quad \texttt{aget}\,\mathit{int}\,u\,v\,(\texttt{and-i}\,(\texttt{not}\,(\texttt{lt}\,v\,\texttt{z})) \\ \qquad\qquad\qquad\qquad\qquad (\texttt{lt}\,v\,u)\,p\,q)\,a\,i') \\ \qquad\qquad (\lambda q{:}\texttt{tr}\,(\texttt{not}\,(\texttt{lt}\,v\,u)).\,10)) \end{array}$$

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

Figure 4: Linear Proof Language Syntax

This code is well-typed (with type *int*), and is therefore safe to execute for any integer i. Although initially nothing is known about the value of i, we can attach an object-level name to that value using `focus`. The outcome of each comparison is then reflected back into the proof system by way of the proof terms introduced by `test` in each branch.

4 Linearity and State

Thus far, we have concerned ourselves with proofs of persistent facts—ones that, once proved, remain true. For example, 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 make all ephemeral facts into linear resources, and craft stateful operations to ensure that they consume any ephemeral facts that they invalidate.

4.1 Linear Proofs

In keeping with our design so far, we use Linear LF [3] as our proof language in the presence of state. To obtain Linear LF, we add the additional constructs in Figure 4 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 may be duplicated or discarded even in Linear LF. 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 refer to linear variables and 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, $A_1 \mathbin{\&} A_2$), 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 function 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, $A_1 \mathbin{\&} A_2$ provides a choice of either A_1 or A_2 but not both. The choice is determined by which projection operation is used. Top 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 collected unused resources.

Note that the linear function type is non-dependent. This is because linear resources are not permitted to be used in families (nor, by implication, in proof kinds, kinds, or constructors). Thus, families, type

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

Figure 5: Linear Programming Language Syntax

constructors, kinds and proof kinds are never scarce resources.

Two types common in linear logic but not provided in Linear LF 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 duplicatible) object, is subsumed by the structure of the language. On the left, the intuitionistic variables are precisely those given by the ordinary (non-linear) context. On the right, an object is considered intuitionistic if and only if it consumes no resources (*i.e.,* it is well-typed in an empty linear context). No facility is provided for functions with types such as $(A \multimap !B)$ that consume resources yet return intuitionistic results.

4.2 Linearity in Programming

The surrounding 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 as in the proof language, as shown in Figure 5.

With the introduction of linearity, terms are typechecked within a linear context, just as objects are. (The new judgement forms are given in Figure 7.) However, there is an important difference between the use of resources in objects and in terms. An object can contain resources, and very often will, even when in canonical form. In term language, however, resources are consumed in process of evaluation, never by constructing a value. This is because the entire purpose of resources is to express ephemeral facts, which may change during evaluation.

This fact leads to some typing rules for terms that are unusual in linear type systems. Two examples serve to summarize. First, values must never consume resources. Since values are freely duplicatible, duplication of a value would duplicate a resource, which we cannot permit. Moreover, values survive as the state changes during evaluation, so resources appearing in values could preserve ephemeral propositions that were no longer true. This means that the type system must insist that values be well-typed only in empty linear contexts, as in the following rule for linear functions:

$$\frac{\Gamma; u\hat{:}A \vdash e : \tau}{\Gamma; \epsilon \vdash \hat{\lambda} u{:}A.e : A \multimap \tau}$$

Second, non-value terms may always consume resources, even though the type system never makes any guarantees regarding the number of uses of a term. This can be seen by considering the previous argument in conjunction with the subject reduction property:[2] by the time terms are reduced to values, they will no longer contain any resources. Thus, function application is typed by the rule

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma; \Delta_2 \vdash e_2 : \tau_1}{\Gamma; (\Delta_1, \Delta_2) \vdash e_1 e_2 : \tau_2}$$

even though e_1 is permitted to use its argument multiple (or zero) times.

Interestingly, these facts 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

[2]Although we have not discussed it yet, the property does in fact hold.

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, as 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.

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

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 $\texttt{apply} : \forall \beta, \gamma{:}T.(\beta \to \gamma) \times \beta \to \gamma$. In LTT, it is essential to be able to provide linear resources to the argument function, otherwise the argument function is crippled for stateful computation. This can be solved using polymorphism over families:

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

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, 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 desirable properties of Linear LF.

To make use of tensor, we add a construct eliminating it to the *term* 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 6 (with signatures omitted for brevity).

The $\texttt{apply}$ 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 $\texttt{map}$ to require some resources to be present, in order to for $\texttt{map}$ to be able to apply that function more than once, $\texttt{map}$ must require that those resources not be consumed. As discussed at the end of the previous

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

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

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

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

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

Figure 6: Tensor

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 7: Linear LTT Judgements

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 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:

$$
\begin{array}{l}
\mathtt{map} : \forall a{:}P^+.\, \forall \beta, \gamma, \delta{:}T. \\
\qquad (\forall \varepsilon{:}T.\, \beta \to (\gamma \to a \multimap \varepsilon) \to a \multimap \varepsilon) \to \mathtt{list}\,\beta \to (\mathtt{list}\,\gamma \to a \multimap \delta) \to a \multimap \delta
\end{array}
$$

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 7. 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 $\mathit{Mlk} \in \mathit{Dom}(R)$, $\vdash_S R(\mathit{Mlk}) : P$.

The full typechecking rules for LTT appear in Appendix A. 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 [17] of decidable typechecking for Linear LF.

As is often the case, typechecking in LTT boils down 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, $\vdash_S S$ context, and $\Gamma \vdash_S \Delta$ 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 [17] is based on a logical relations argument modeled after the analogous proof of Harper and Pfenning [8] for intuitionistic LF. From this it is easy to show decidability of LTT typechecking:

Corollary 4.3 *Suppose S and R are well-formed, $\vdash_S S$ context, and $\Gamma \vdash_S \Delta$ 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 Crary *et al.* [5].

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, $\mathit{valid}\, C$ 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 $\mathit{ref}\,\tau\, C$, where τ is the type of the cell's contents.

The operations available on reference cells have the following types:

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

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 regardless of the contents of the linear context.

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 Modular Semantics

We are now ready to begin discussing the operational semantics of LTT. The operational semantics consists of two parts: a small operational semantics for the built-in constructs of LTT, and the semantics of whatever term constants are supplied by the signature. As discussed earlier, it is advantageous for ease of scalability for the semantics of term constants to be definable in a modular fashion, so that independent constructs can independently be given semantics and those semantics proven sound, without any need for an overall soundness proof. We present such a framework for modular semantics here.

A semantic module consists of two parts, a static module and a dynamic module. A static module specifies the set of constants that the module supplies, and a dynamic module gives the transition rules implementing the constants of a static module. We show that if two static modules are compatible (in the sense that they provide different constants) then they can be safely joined together, and that two dynamic modules implementing them can be joined together to form a single dynamic module implementing the join of the static modules. A full operational semantics is then assembled simply by joining together dynamic modules implementing compatible static modules.

Before we make this rigorous, we require some preliminary results regarding operational semantics. Then we proceed to formalize the notion of static modules and then dynamic modules. We will ignore state for the time being, but pick it up again in Section 6.4.

6.1 Operational Semantics

Core evaluation (that is, evaluation of built-in LTT constructs) is defined as in Figure 8. We also define a transition system to be a mapping from terms to terms. We intend transition systems to account for the semantics of terms not covered by core evaluation. Given a transition system, we can define a semantics for LTT:

Definition 6.1 Suppose T is a transition system and let values and evaluation contexts be defined as in Figure 9. Then $E[e] \mapsto_T E[e']$ if and only if either $e \mapsto e'$ or $T(e) = e'$. A term e is *T-stuck* if it is not a value and there exists no e' such that $e \mapsto_T e'$.

In order to prove any nice properties about this semantics, we will need to constrain the transition system:

Definition 6.2 Suppose S is a well-formed signature, Z is a set of term constants and T is a transition system. Let frames be defined as in Figure 10. Then *T implements Z relative to S* if:

- for all $e \in Dom(T)$, e is of the form $F[ek]$, for some $ek \in Dom(Z)$, and
- for all frames F, $ek \in Z$, and types τ, if $\vdash_S F[ek] : \tau$, then $F[ek] \in Dom(T)$ and $\vdash_S T(F[ek]) : \tau$.

The first clause will be helpful later on in showing determinacy when we join transition systems together (Lemma 6.5); it says essentially that transition systems implementing disjoint sets of constants do not conflict. The second clause essentially provides the Progress and Subject Reduction properties; it says that

$$
\begin{array}{lcl}
(\lambda x{:}\tau.e)v & \mapsto & e[v/x] \\
\mathsf{let}\ \langle x_1, x_2\rangle = \langle v_1, v_2\rangle\ \mathsf{in}\ e & \mapsto & e[v_1, v_2/x_1, x_2] \\
(\lambda\alpha{:}k.e)c & \mapsto & e[c/\alpha] \\
\mathsf{let}\ \langle \alpha, x\rangle = (\mathsf{pack}\ \langle c, v\rangle\ \mathsf{as}\ \ldots)\ \mathsf{in}\ e & \mapsto & e[c, v/\alpha, x] \\
(\lambda a{:}K.e)A & \mapsto & e[A/a] \\
\mathsf{let}\ \langle a, x\rangle = (\mathsf{pack}\ \langle A, v\rangle\ \mathsf{as}\ \ldots)\ \mathsf{in}\ e & \mapsto & e[A, v/a, x] \\
(\lambda u{:}A.e)M & \mapsto & e[M/u] \\
\mathsf{let}\ \langle u, x\rangle = (\mathsf{pack}\ \langle M, v\rangle\ \mathsf{as}\ \ldots)\ \mathsf{in}\ e & \mapsto & e[M, v/u, x] \\
(\lambda u{:}A.e)\hat{\ }M & \mapsto & e[M/u] \\
\mathsf{let}\ \langle\!\langle u_1, u_2\rangle\!\rangle = \langle\!\langle M_1, M_2\rangle\!\rangle\ \mathsf{in}\ e & \mapsto & e[M_1, M_2/u_1, u_2] \\
\mathsf{let}\ \star = \star\ \mathsf{in}\ e & \mapsto & e
\end{array}
$$

Figure 8: Core Evaluation

$$
\begin{array}{llcl}
\textit{values} & v & ::= & x \mid ek \mid \lambda x{:}c.e \mid \langle v_1, v_2\rangle \mid \\
 & & & \lambda\alpha{:}k.e \mid \mathsf{pack}\ \langle c, v\rangle\ \mathsf{as}\ \Sigma\alpha{:}k.c \mid \\
 & & & \lambda a{:}K.e \mid \mathsf{pack}\ \langle A, v\rangle\ \mathsf{as}\ \Sigma a{:}K.c \mid \\
 & & & \lambda u{:}A.e \mid \mathsf{pack}\ \langle M, v\rangle\ \mathsf{as}\ \Sigma u{:}A.c \mid \\
 & & & \hat{\lambda} u{:}A.e \\
\textit{eval contexts} & E & ::= & [\,] \mid E\,e \mid v\,E \mid \pi_1 E \mid \pi_2 E \mid \\
 & & & E\,c \mid \mathsf{let}\ \langle \alpha, x\rangle = E\ \mathsf{in}\ e \mid \\
 & & & E\,A \mid \mathsf{let}\ \langle a, x\rangle = E\ \mathsf{in}\ e \mid \\
 & & & E\,M \mid \mathsf{let}\ \langle u, x\rangle = E\ \mathsf{in}\ e \mid \\
 & & & E\hat{\ }M
\end{array}
$$

Figure 9: Values and Evaluation Contexts

$$
\begin{array}{llcl}
\textit{frames} & F & ::= & [\,]\,v \mid \mathsf{let}\ \langle x_1, x_2\rangle = [\,]\ \mathsf{in}\ e \mid \\
 & & & [\,]\,c \mid \mathsf{let}\ \langle \alpha, x\rangle = [\,]\ \mathsf{in}\ e \mid \\
 & & & [\,]\,A \mid \mathsf{let}\ \langle a, x\rangle = [\,]\ \mathsf{in}\ e \mid \\
 & & & [\,]\,M \mid \mathsf{let}\ \langle u, x\rangle = [\,]\ \mathsf{in}\ e \mid \\
 & & & [\,]\hat{\ }M
\end{array}
$$

Figure 10: Evaluation Frames

for any well-typed operation on a constant being implemented (where frames specify operations), a transition is defined and its result is well-typed.

We will say that T implements S if it implements some Z containing the term constants of $Dom(S)$ (relative to S). When T implements S, then we enjoy the type safety property:

Lemma 6.3 (Type Safety) *Suppose T implements S. Then:*

- **(Determinism)** *If $e \mapsto_T e'$ and $e \mapsto_T e''$ then $e' = e''$.*
- **(Subject reduction)** *If $\vdash_S e : \tau$ and $e \mapsto_T e'$ then $\vdash_S e' : \tau$.*
- **(Progress)** *If $\vdash_S e : \tau$ then e is not T-stuck.*

Finally, we can define what it means to join two transition systems together and when it is permissible to do so:

Definition 6.4 Suppose T_1 and T_2 have disjoint domains. Then the join of T_1 and T_2 (written $T_1 \bowtie T_2$) is $T_1 \uplus T_2$.

Lemma 6.5 *If T_1 implements Z_1 relative to S, T_2 implements Z_2 relative to S, and Z_1 and Z_2 are disjoint, then $T_1 \bowtie T_2$ is well-defined and implements $Z_1 \uplus Z_2$ relative to S.*

6.2 Static Modules

A static module must specify the complete set of constants necessary to compute with the module, including any term values. For example, a static module for arrays must supply a term constant for every possible array. However, in the presence of modularity, it cannot be clear *a priori* what constants are necessary, since values can be built from values from other modules, as yet unknown. Thus, we define a static module to have two parts: a termless portion, and a function that provides all the necessary term constants given a signature that summarizes all the modules being included. Later, we will extract the final signature from a collection of static modules using a fixed point construction.

Definition 6.6 A *static module SM* is a pair (S, φ) where:

- S is a termless signature (*i.e.,* a signature providing no term constants), and
- φ is a function from signatures to term signatures (*i.e.,* signatures providing only term constants).

Definition 6.7 A static module (S, φ) is well-formed if:

- S is well-formed, and
- for all $S' \supseteq S$, if S' is well-formed then $\varphi(S') \uplus \mathit{TLP}(S')$ is well-formed, and
- φ is monotone, and
- for all S' and $ek \in Dom(\varphi(S'))$, either:
  - $\varphi(S')(ek)$ has the form $ck\, E_1 \cdots E_n$ for some $ck \in Dom(S)$, or
  - the outermost operator of $\varphi(S')(ek)$ is $\rightarrow$, $\times$, Π, Σ, or $\multimap$.

The first two clauses specify basic well-formedness conditions for a static module. The third ensures that the eventual fixed point construction will work. The fourth ensures the existence of useful canonical forms by prohibiting static modules from adding new constants to the types of other modules. We will exploit this clause in Lemma 6.20.

Next we can define compatibility and the joining of static modules:

Definition 6.8 Static modules (S_1, φ_1) and (S_2, φ_2) are *compatible* if S_1 and S_2 have disjoint domains, and for all S, $\varphi_1(S)$ and $\varphi_2(S)$ have disjoint domains.

$$
\varphi_{\text{array}}(S) = \left\{ \begin{array}{lll}
\texttt{mkarray} & : \Pi\alpha{:}T.\Pi u{:}\mathit{Int}. & \\
 & \quad \mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,u\,\texttt{z})) \to & \\
 & \quad S_{\mathit{Int}}\,u \to \alpha \to \mathit{array}\,u\,\alpha & \\
\texttt{mkarray}_{\tau} & : \Pi u{:}\mathit{Int}. & \\
 & \quad \mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,u\,\texttt{z})) \to & \\
 & \quad S_{\mathit{Int}}\,u \to \tau \to \mathit{array}\,u\,\tau & (\text{for all } \vdash_S \tau : T) \\
\texttt{mkarray}_{\tau,M} & : \mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,M\,\texttt{z})) \to & \\
 & \quad S_{\mathit{Int}}\,M \to \tau \to \mathit{array}\,M\,\tau & (\text{for all } \vdash_S \tau : T,\, \vdash_S M : \mathit{Int}) \\
\texttt{mkarray}_{\tau,M,M'} & : S_{\mathit{Int}}\,M \to \tau \to \mathit{array}\,M\,\tau & (\text{for all } \vdash_S \tau : T,\, \vdash_S M : \mathit{Int},\, \vdash_S M' : \mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,M\,\texttt{z}))) \\
\texttt{mkarray}_{\tau,M,M',v} & : \tau \to \mathit{array}\,M\,\tau & (\text{for all } \vdash_S \tau : T,\, \vdash_S M : \mathit{Int},\, \vdash_S M' : \mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,M\,\texttt{z})) \\
 & & \qquad \vdash_S v : S_{\mathit{Int}}\,M) \\
[v_1, \ldots, v_n] & : \mathit{array}\,(\texttt{s}^n\texttt{z})\,\tau & (\text{for all } n \geq 0,\, \vdash_S \tau : T,\, \vdash_S v_i : \tau) \\
n & : \mathit{int} & (\text{for all integers } n) \\
\overline{n} & : S_{\mathit{Int}}(\texttt{s}^n\texttt{z}) & (\text{for all } n \geq 0) \\
\overline{n} & : S_{\mathit{Int}}(\texttt{neg}(\texttt{s}^{-n}\texttt{z})) & (\text{for all } n < 0) \\
\vdots & &
\end{array} \right\}
$$

Figure 11: A Static Module for Arrays

Definition 6.9 Suppose $SM_1 = (S_1, \varphi_1)$ and $SM_2 = (S_2, \varphi_2)$ are compatible. Then the *join* of SM_1 and SM_2 (written $SM_1 \bowtie SM_2$) is $(S_1 \uplus S_2, \varphi_1 \uplus \varphi_2)$, where the disjoint union of functions is computed pointwise.

Lemma 6.10 *Joining of static modules is associative, commutative, and has a unit.*

Lemma 6.11 *If SM_1 and SM_2 are well-formed and compatible, then $SM_1 \bowtie SM_2$ is well-formed.*

Finally we can define the fixed point construction that extracts a signature from a static module:

Definition 6.12 Suppose $SM = (S, \varphi)$ is a static module with φ monotone. Then the fixed point of SM (written $\mathit{fix}(SM)$) is the least fixed point of f, the monotone operator on signatures defined by:

$$f(S') = S \uplus \varphi(S')$$

Lemma 6.13 *If SM is well-formed, then $\mathit{fix}(SM)$ is a well-formed signature.*

Proof

Let $SM = (S, \varphi)$ and let f be defined as above. Observe that if S' is well-formed and its typeless portion is S then $f(S')$ has the same property. Also, the property holds for $f(\emptyset)$, since $\varphi(\emptyset)$ is contained in $\varphi(S)$, which is well-formed (when combined with S). Finally, the property is preserved by unions of chains. The result follows by transfinite induction.

We conclude with a definition that will be convenient in the next section:

Definition 6.14 Suppose SM is well-formed. Then S completes SM (written $S \triangleright SM$) if there exists a well-formed SM' compatible with SM such that $S = \mathit{fix}(SM \bowtie SM')$.

Example To make this concrete, we give a static module corresponding to the arrays of Section 3. The non-term constants are exactly those given in Section 3, so we will devote our attention to the term portion of the static module, φ_{array}. Our static module, given a signature, must return three sort of term constants: the actual term constants from Section 3, partially evaluated forms of those constants, and value forms for each type the module provides (in this example, *array*, *int*, S_{int}, and S_o). We give an illustrative fragment in Figure 11. Now let $SM_{\text{array}} = (S_{\text{array}}, \varphi_{\text{array}})$, where S_{array} contains the non-term constants from Section 3. It is easy to check that all the conditions for well-formedness are satisfied.

6.3 Dynamic Modules

A dynamic module will be defined to be a function from signatures to transitions systems, the idea being that it is applied to a fixed point built over its static module, and returns a transition system implementing all the term constants its static module provides.

Definition 6.15 A *dynamic module* DM is a function from signatures to transition systems.

Definition 6.16 Suppose $SM = (S, \varphi)$ is well-formed. Then DM implements SM if for all $S' \rhd SM$, $DM(S')$ implements $Dom(\varphi(S'))$ relative to S'.

When dynamic modules implement compatible static modules, they can be joined together:

Definition 6.17 Suppose that for all S, $DM_1(S)$ and $DM_2(S)$ have disjoint domains. Then the *join* of DM_1 and DM_2 (written $DM_1 \bowtie DM_2$) is the pointwise join of DM_1 and DM_2.

Lemma 6.18 *If DM_1 implements SM_1, DM_2 implements SM_2, and SM_1 and SM_2 are compatible, then $DM_1 \bowtie DM_2$ is well-defined and implements $SM_1 \bowtie SM_2$.*

Proof

Let $SM_1 = (S_1, \varphi_1)$ and $SM_2 = (S_2, \varphi_2)$. Suppose SM is well-formed and compatible with $SM_1 \bowtie SM_2$, and let $S' = \mathit{fix}(SM_1 \bowtie SM_2 \bowtie SM)$. Then SM_1 is compatible with $SM_2 \bowtie SM$ so $S' \rhd SM_1$. Thus $DM_1(S')$ implements $Dom(\varphi_1(S'))$ relative to S'. Similarly $DM_2(S')$ implements $Dom(\varphi_2(S'))$ relative to S'. Hence, by Lemma 6.5, $DM_1(S') \bowtie DM_2(S')$ implements $Dom(\varphi_1(S')) \uplus Dom(\varphi_2(S'))$ relative to S'.

Finally, a dynamic module applied to its static module's fixed point, implements that fixed point (and consequently the type safety theorem applies):

Lemma 6.19 *If DM implements SM then $DM(\mathit{fix}(SM))$ implements $\mathit{fix}(SM)$.*

Proof

Let $SM = (S, \varphi)$ and $S' = \mathit{fix}(SM)$. Observe that $S' \rhd SM$. Then $DM(S')$ implements $Dom(\varphi(S'))$ relative to S'. Since $S' = S \uplus \varphi(S')$, the term constants of S' are exactly those of $\varphi(S')$. Hence $DM(S')$ implements S'.

With these tools established, the primary proof burden in using modular semantics is to prove the dynamic module's implementation theorem, which corresponds to the usual type safety theorem in non-modular semantics. The following lemma is very useful in proving such implementation theorems. It insulates the implementation theorem from the gory details of completion and the fixed point construction, thereby making the proof little more complicated than the usual non-modular safety proof.

Lemma 6.20 (Canonical Forms) *Suppose $SM = (S, \varphi)$ and $S' \rhd SM$. Then:*

- *If $\vdash_{S'} ek : ck\, E_1 \cdots E_n$ for some $ck \in Dom(S)$, then $ek \in Dom(\varphi(S'))$.*
- *If $\vdash_{S'} Mk : A$ where A is canonical and contains some $Ak \in Dom(S)$, then $Mk \in Dom(S)$ (and consequently $Mk \notin Dom(S') \setminus Dom(S)$).*

Proof

We show the first part, the second is similar. By construction, $S'(ek)$ has the form $ck'\, E'_1 \cdots E'_m$, and by inspection of the typing rules, $\vdash_{S'} S'(ek) = ck\, E_1 \cdots E_n$. Thus $ck = ck'$ and $m = n$. By definition of completion, $S' = S \uplus S_2 \uplus \varphi(S') \uplus \varphi_2(S')$, for some well-formed (S_2, φ_2) compatible with SM. Suppose $ek \in Dom(\varphi_2(S'))$. Since (S_2, φ_2) is well-formed, it follows that $ck \in Dom(S_2)$. However, S_2 is disjoint from S (by compatibility), so ek must lie in $Dom(\varphi(S'))$ instead.

Let $DM_{\text{array}}(S) = T_S$, where T_S is given by:

$$\begin{array}{lcl} T_S(\texttt{mkarray}\,\tau) & = & \texttt{mkarray}_\tau \\ T_S(\texttt{mkarray}_\tau\,M) & = & \texttt{mkarray}_{\tau,M} \\ \vdots & & \\ T_S(\texttt{mkarray}_{\tau,M,M',v}\,v') & = & [\overbrace{v',\ldots,v'}^{n}] \quad (\text{if } v = \overline{n}) \\ \vdots & & \end{array}$$

(for $\texttt{mkarray}_\tau, \texttt{mkarray}_{\tau,M,M',v}, \ldots \in \mathit{Dom}(\varphi_{\text{array}}(S))$)

Figure 12: A Dynamic Module for Arrays

Example To make this concrete, we give a dynamic module implementing the static module for arrays from the previous section. An illustrative fragment is given in Figure 12.

It remains to show that DM_{array} implements SM_{array}. Suppose $S \rhd SM_{\text{array}}$. Note that, by construction, $S_{\text{array}} \subseteq S$. We wish to show that $DM(S)$ implements Z relative to S, where $Z = \mathit{Dom}(\varphi_{\text{array}}(S)) = \{\texttt{mkarray}, \texttt{mkarray}_\tau, \ldots\}$. It is easy to show that the first clause of implementation is satisfied.

For the second, suppose $\vdash_S F[\mathit{ek}] : \tau$ for some ek in Z. We wish to show that $T_S(F[\mathit{ek}])$ is defined and $\vdash_S T_S(F[\mathit{ek}]) : \tau$. We proceed by case analysis on ek.

Suppose ek is $\texttt{mkarray}$. Since $F[\mathit{ek}]$ is well-typed, F must be of the form $[\,]c$ where $\vdash_S c : T$. Then $T_S(\texttt{mkarray}\,c) = \texttt{mkarray}_c$, and $\texttt{mkarray}_c$ has the desired type. Similar reasoning applies to all the partially evaluated forms of $\texttt{mkarray}$ except the last.

The interesting case is when ek is $\texttt{mkarray}_{\tau,M,M',v}$. Since $F[\mathit{ek}]$ is well-typed, F must be of the form $[\,]v'$ where $\vdash_S v' : \tau$. We wish to show that (for some n) $v = \overline{n}$ and $n \geq 0$. Since $\texttt{mkarray}_{\tau,M,M',v} \in \mathit{Dom}(\varphi_{\text{array}}(S))$, we know that $\vdash_S M : \mathit{Int}$ and $\vdash_S M' : \mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,M\,\texttt{z}))$ and $\vdash_S v : S_{\mathit{Int}}\,M$.

Certainly v must be an object constant, and by Lemma 6.20, it must be drawn from $\varphi_{\text{array}}(S)$. Thus $v = \overline{n}$. Suppose $n < 0$. Then, since $\overline{n} : S_{\mathit{Int}}\,M$, it follows that $\vdash_S M = \texttt{neg}(\texttt{s}^{-n}\texttt{z}) : \mathit{Int}$. Hence $\vdash_S M' : \mathit{tr}\,(\texttt{not}\,(\texttt{lt}\,(\texttt{neg}(\texttt{s}^{-n}\texttt{z}))\,\texttt{z}))$. Again by Lemma 6.20, M' must be built exclusively from object constants appearing in S_{array}. Therefore, assuming S_{array} contains only appropriate proof terms, we may prove by induction over the structure of M' that M' represents a valid proof of $\neg(-(-n) < 0)$. (This fact is analogous to the usual adequacy theorem used in LF [7].) Since no such proof can exist, $n \geq 0$.

Thus $T_S(F[\mathit{ek}]) = [v', \ldots, v']$. It remains only to show that $\vdash_S [v', \ldots, v'] : \mathit{array}\,M\,\tau$, and this holds provided that $\vdash_S \texttt{s}^n\texttt{z} = M : \mathit{Int}$. The latter follows from $\vdash_S \overline{n} : S_{\mathit{Int}}\,M$.

6.4 Modular Semantics and State

Since a particular notion of state (or lack thereof) is often central to the design of a programming language, it is not clear that being able to modularize a stateful semantics is as profitable as for a stateless semantics. Moreover, it appears that generalizing the preceding framework to account for state complicates it severely. Instead, we intend that users of LTT construct a particular operational semantics (depending on their desired notion of state), but predicate that construction over an unspecified collection of semantic modules.

We will not construct a full example here; instead we summarize the key points. First, one defines a notion of a type for the state. (For example, the heap types of Typed Assembly Language.)

Second, one defines a static module parametrized over state types. Its termless portion is invariant, but its term portion uses the heap type to determine what references to the state are permissible. (In TAL's case, the permissible references are the currently allocated pointers, as indicated by the state type.) Very often state types will contain ordinary types, so well-formedness of a state type cannot be defined independently of a signature, and therefore the static module cannot assume its argument state type to be well-formed. However, it is not difficult to define that static module so that any ill-formed components of the state type are ignored.

Third, one defines a function (call it $\hat{R}$) from state types to linear signatures. This provides resources to the program that are usable only for the current state, and need not be monotone. (There is no analog of this in TAL, but for the memory management language of Section 5, state types would track which cells had been deallocated, and the function would provide linear constants for every valid cell.)

Finally, the static module is joined with some other static module SM and their fixed point is taken to produce a signature. The resulting signature (call it $\hat{S}$) is parametrized over state types, since it is built from a parametrized static module. The two parametrized signatures $\hat{S}$ and $\hat{R}$ are used to type machine configurations using a rule like

$$\frac{\vdash \sigma : \Psi \quad \vdash_{\hat{S}(\Psi),\hat{R}(\Psi)} e : \tau}{\vdash_{\hat{S},\hat{R}} (\sigma, e) : \tau}$$

where σ ranges over states and Ψ ranges over state types.

For the operational semantics, a stateful transition system can be constructed deferring to the dynamic module implementing SM for any unknown components. In the safety proof we take advantage of the above constructions as follows: When a linear object constant appears in a well-typed term, then it must be supplied by $\hat{R}(\sigma)$, which provides information about the current state and may show that an operation being performed is permissible. Similar information is gleaned from appearances of term constants from $\hat{S}(\sigma)$.

7 Conclusion

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 current 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 Crary *et al.* [5] 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 propositions (represented equivalences and constraints) and proof terms. Beyond this, we conjecture that the security automata type system of Walker [19] and the alias-tracking type system of Smith *et al.* [16] can easily be encoded in LTT as well.

The casting of all these type systems into one uniform framework by itself promotes one sort of re-use, that of the typechecking software. In order to be able to re-use safety proofs as well (which can be just as burdensome to produce as typechecking software), we provide a theory for modular development of operational semantics and the safety proofs thereof. To our knowledge, this is the only such account.

A similar effort to LTT, as far as its re-usability goals are concerned, is the TinkerType meta-language of Levin and Pierce [10]. 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.*). TinkerType provides for modular development of typechecking software, but not of safety proofs. However, TinkerType's main feature is support for inheritance; for example $F_{\leq}$ inherits from F and adds subtyping. LTT makes no effort at supporting inheritance; each semantic module to be combined must be fully developed on its own. Adding this is an interesting avenue for future work.

Acknowledgements

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

A Typing Rules

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 $Dom(\Gamma)$ or $Dom(\Delta)$. A similar side-condition applies in rules with only an intuitionistic context.

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

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

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

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

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

$$\frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S k \text{ kind}}{\Gamma \vdash_S \Pi u{:}A.k \text{ kind}}$$

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

$$\frac{}{\Gamma \vdash_S T = T \text{ kind}} \qquad \frac{\Gamma \vdash_S k_1 = k_2 \text{ kind} \quad \Gamma \vdash_S k_1' = k_2' \text{ kind}}{\Gamma \vdash_S k_1 \to k_1' = k_2 \to k_2' \text{ kind}} \qquad \frac{\Gamma \vdash_S K_1 = K_2 \text{ pkind} \quad \Gamma, a{:}K_1 \vdash_S k_1 = k_2 \text{ kind}}{\Gamma \vdash_S \Pi a{:}K_1.k_1 = \Pi a{:}K_2.k_2 \text{ kind}}$$

$$\frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma, u{:}A_1 \vdash_S k_1 = k_2 \text{ kind}}{\Gamma \vdash_S \Pi u{:}A_1.k_1 = \Pi u{:}A_2.k_2 \text{ kind}} \qquad \frac{\Gamma \vdash_S k \text{ kind}}{\Gamma \vdash_S k = k \text{ kind}} \qquad \frac{\Gamma \vdash_S k_2 = k_1 \text{ kind}}{\Gamma \vdash_S k_1 = k_2 \text{ kind}}$$

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

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

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

$$\frac{\Gamma \vdash_S c_1 : k_1 \to k_2 \quad \Gamma \vdash_S c_2 : k_1}{\Gamma \vdash_S c_1 c_2 : k_2} \qquad \frac{\Gamma \vdash_S K \text{ pkind} \quad \Gamma, a{:}K \vdash_S c : k}{\Gamma \vdash_S \lambda a{:}K.c : \Pi a{:}K.k} \qquad \frac{\Gamma \vdash_S c : \Pi a{:}K.k \quad \Gamma \vdash_S A : K}{\Gamma \vdash_S c\, A : k[A/a]}$$

$$\frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S c : k}{\Gamma \vdash_S \lambda u{:}A.c : \Pi u{:}A.k} \qquad \frac{\Gamma \vdash_S c : \Pi u{:}A.k \quad \Gamma; \epsilon \vdash_{S,\emptyset} M : A}{\Gamma \vdash_S c\, M : k[M/u]} \qquad \frac{\Gamma \vdash_S \tau_1 : T \quad \Gamma \vdash_S \tau_2 : T}{\Gamma \vdash_S \tau_1 \to \tau_2 : T}$$

$$\frac{\Gamma \vdash_S \tau_1 : T \quad \Gamma \vdash_S \tau_2 : T}{\Gamma \vdash_S \tau_1 \times \tau_2 : T} \qquad \frac{\Gamma \vdash_S k \text{ kind} \quad \Gamma, \alpha{:}k \vdash \tau : T}{\Gamma \vdash_S \Pi\alpha{:}k.\tau : T} \qquad \frac{\Gamma \vdash_S k \text{ kind} \quad \Gamma, \alpha{:}k \vdash \tau : T}{\Gamma \vdash_S \Sigma\alpha{:}k.\tau : T}$$

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

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

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

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

$$\frac{\Gamma \vdash_S c_1 = c_2 : k_1 \to k_2 \quad \Gamma \vdash_S c_1' = c_2' : k_1}{\Gamma \vdash_S c_1\, c_1' = c_2\, c_2' : k_2} \qquad \frac{\begin{array}{c}\Gamma, a{:}K \vdash_S c_1 = c_2 : k \\ \Gamma \vdash_S K_1 = K \text{ pkind} \quad \Gamma \vdash_S K_2' = K \text{ pkind}\end{array}}{\Gamma \vdash_S \lambda\alpha{:}K_1.c_1 = \lambda\alpha{:}K_2.c_2 : \Pi a{:}K.k}$$

$$\frac{\Gamma \vdash_S c_1 = c_2 : \Pi a{:}K.k \quad \Gamma \vdash_S A_1 = A_2 : K}{\Gamma \vdash_S c_1\, A_1 = c_2\, A_2 : k[A_1/a]} \qquad \frac{\begin{array}{c}\Gamma, u{:}A \vdash_S c_1 = c_2 : k \\ \Gamma \vdash_S A_1 = A : P \quad \Gamma \vdash_S A_2 = A : P\end{array}}{\Gamma \vdash_S \lambda u{:}A_1.c_1 = \lambda u{:}A_2.c_2 : \Pi u{:}A.k}$$

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

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

$$\frac{\Gamma \vdash_S k_1 = k_2 \ \mathsf{kind} \quad \Gamma, \alpha{:}k_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Sigma\alpha{:}k_1.\tau_1 = \Sigma\alpha{:}k_2.\tau_2 : T} \qquad \frac{\Gamma \vdash_S K_1 = K_2 \ \mathsf{pkind} \quad \Gamma, a{:}K_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Pi a{:}K_1.\tau_1 = \Pi a{:}K_2.\tau_2 : T}$$

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

$$\frac{\Gamma \vdash_S A_1 = A_2 : P \quad \Gamma, u{:}A_1 \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S \Sigma u{:}A_1.\tau_1 = \Sigma u{:}A_2.\tau_2 : T} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : P^+ \quad \Gamma \vdash_S \tau_1 = \tau_2 : T}{\Gamma \vdash_S A_1 \multimap \tau_1 = A_2 \multimap \tau_2 : T}$$

$$\frac{\Gamma \vdash_S c_1 = c_2 : k \quad \Gamma \vdash_S k = k' \ \mathsf{kind}}{\Gamma \vdash_S c_1 = c_2 : k'} \qquad \frac{\Gamma, \alpha{:}k_1 \vdash_S c_1 : k_2 \quad \Gamma \vdash_S c_2 : k_1}{\Gamma \vdash_S (\lambda\alpha{:}k_1.c_1)\, c_2 = c_1[c_2/\alpha] : k_2}$$

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

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

$$\frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A \vdash_S c_1\, u = c_2\, u : k}{\Gamma \vdash_S c_1 = c_2 : \Pi u{:}A.k} \qquad \frac{\Gamma \vdash_S c : k}{\Gamma \vdash_S c = c : k}$$

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

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

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

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

$$\frac{\Gamma; \Delta \vdash_{S,r} e : \tau_1 \times \tau_2}{\Gamma; \Delta \vdash_{S,R} \pi_i e : \tau_i}\ (i = 1, 2) \qquad \frac{\Gamma \vdash_S k \ \mathsf{kind} \quad \Gamma, \alpha{:}k; \epsilon \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda\alpha{:}k.e : \Pi\alpha{:}k.\tau}$$

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

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \Sigma\alpha{:}k.\tau \\ (\Gamma, \alpha{:}k, x{:}\tau); \Delta_2 \vdash_{S,R_2} e_2 : \tau'\end{array}}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \mathsf{let}\ \langle \alpha, x \rangle = e_1\ \mathsf{in}\ e_2 : \tau'} \qquad \frac{\Gamma \vdash_S K \ \mathsf{pkind} \quad \Gamma, a{:}K; \epsilon \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda a{:}K.e : \Pi a{:}K.\tau}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} e : \Pi a{:}K.\tau \quad \Gamma \vdash_S A : K}{\Gamma; \Delta \vdash_{S,R} e\, A : \tau[A/a]} \qquad \frac{\Gamma \vdash_S A : K \quad \Gamma, a{:}K \vdash_S \tau : T \quad \Gamma; \Delta \vdash_{S,R} e : \tau[A/a]}{\Gamma; \Delta \vdash_{S,R} \mathsf{pack}\ \langle A, e \rangle\ \mathsf{as}\ \Sigma a{:}K.\tau : \Sigma a{:}K.\tau}$$

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \Sigma a{:}K.\tau \\ (\Gamma, a{:}K, x{:}\tau); \Delta_2 \vdash_{S,R_2} e_2 : \tau'\end{array}}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \mathsf{let}\ \langle a, x \rangle = e_1\ \mathsf{in}\ e_2 : \tau'} \qquad \frac{\Gamma \vdash_S A : P \quad \Gamma, u{:}A; \epsilon \vdash_{S,\emptyset} e : \tau}{\Gamma; \epsilon \vdash_{S,\emptyset} \lambda u{:}A.e : \Pi u{:}A.\tau}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} e : \Pi u{:}A.\tau \quad \Gamma; \epsilon \vdash_{S,\emptyset} M : A}{\Gamma; \Delta \vdash_{S,R} e\, M : \tau[M/u]} \qquad \frac{\begin{array}{c}\Gamma; \epsilon \vdash_{S,\emptyset} M : A \\ \Gamma, u{:}A \vdash_S \tau : T \quad \Gamma; \Delta \vdash_{S,R} e : \tau[M/u]\end{array}}{\Gamma; \Delta \vdash_{S,R} \mathsf{pack}\ \langle M, e \rangle\ \mathsf{as}\ \Sigma u{:}A.\tau : \Sigma u{:}A.\tau}$$

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash_{S,R_1} e_1 : \Sigma u{:}A.\tau \\ (\Gamma, u{:}A, x{:}\tau); \Delta_2 \vdash_{S,R_2} e_2 : \tau'\end{array}}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \mathsf{let}\ \langle u, x \rangle = e_1\ \mathsf{in}\ e_2 : \tau'} \qquad \frac{\begin{array}{c}\Gamma \vdash_S A : P^+ \\ \Gamma; \Delta, u\hat{:}A \vdash_{S,\emptyset} e : \tau\end{array}}{\Gamma; \epsilon \vdash_{S,\emptyset} \hat{\lambda} u{:}A.e : A \multimap \tau} \qquad \frac{\begin{array}{c}\Gamma; \Delta_1 \vdash_{S,R_1} e : A \multimap \tau \\ \Gamma; \Delta_2 \vdash_{S,R_2} M : A\end{array}}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} e\,\hat{}\,M : \tau}$$

$$\frac{\begin{array}{c}\Gamma; \Delta_1 \vdash_{S,R_1} M : A_1 \otimes A_2 \\ \Gamma; (\Delta_2, u_1\hat{:}A_1, u_2\hat{:}A_2) \vdash_{S,R_2} e : \tau\end{array}}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \mathsf{let}\ \langle\langle u_1, u_2 \rangle\rangle = M\ \mathsf{in}\ e : \tau} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} M : 1 \quad \Gamma; \Delta_2 \vdash_{S,R_2} e : \tau}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} \mathsf{let}\ \star = M\ \mathsf{in}\ e : \tau}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} e : \tau \quad \Gamma \vdash_S \tau = \tau' : T}{\Gamma; \Delta \vdash_{S,R} e : \tau'}$$

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

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

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

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

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

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

$$\frac{}{\Gamma \vdash_S a : K}\ (\Gamma(a) = K) \qquad \frac{}{\Gamma \vdash_S Ak : K}\ (S(Ak) = K) \qquad \frac{\Gamma \vdash_S A_1 : P \quad \Gamma, u{:}A_1 \vdash_S A_2 : K}{\Gamma \vdash_S \lambda u{:}A_1.A_2 : \Pi u{:}A_1.K}$$

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

$$\frac{\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P}{\Gamma \vdash_S A_1 \mathbin{\&} A_2 : P} \qquad \frac{}{\Gamma \vdash_S \top : P} \qquad \frac{\Gamma \vdash_S A_1 : P^+ \quad \Gamma \vdash_S A_2 : P^+}{\Gamma \vdash_S A_1 \otimes A_2 : P^+} \qquad \frac{}{\Gamma \vdash_S 1 : P^+}$$

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

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

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

$$\frac{\begin{array}{c}\Gamma, u{:}A \vdash_S A_1' = A_2' : K \\ \Gamma \vdash_S A : P \quad \Gamma \vdash_S A_1 = A : P \quad \Gamma \vdash_S A_2 = A : P\end{array}}{\Gamma \vdash_S \lambda u{:}A_1.A_1' = \lambda u{:}A_2.A_2' : \Pi u{:}A.K} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : \Pi u : B.K \quad \Gamma \vdash_S M_1 = M_2 : B}{\Gamma \vdash_S A_1\, M_1 = A_2\, M_2 : K[M_1/u]}$$

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

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

$$\frac{\Gamma \vdash_S A : K}{\Gamma \vdash_S A = A : K} \qquad \frac{\Gamma \vdash_S A_2 = A_1 : K}{\Gamma \vdash_S A_1 = A_2 : K} \qquad \frac{\Gamma \vdash_S A_1 = A_2 : K \quad \Gamma \vdash_S A_2 = A_3 : K}{\Gamma \vdash_S A_1 = A_3 : K}$$

$$\frac{\begin{array}{c}\Gamma \vdash_S B : P \\ \Gamma, u{:}B \vdash_S A_1 = A_2 : K \quad \Gamma; \epsilon \vdash_S M_1 = M_2 : B\end{array}}{\Gamma \vdash_S (\lambda u{:}B.A_1)\, M_1 = A_2[M_2/u] : K[M_1/u]} \qquad \frac{\begin{array}{c}\Gamma \vdash_S B : P \\ \Gamma, u{:}B \vdash_S A_1\, u = A_2\, u : K\end{array}}{\Gamma \vdash_S A_1 = A_2 : \Pi u{:}B.K}$$

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

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

$$\frac{}{\Gamma; \epsilon \vdash_{S,\{Mlk:A\}} Mlk : A} \qquad \frac{\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P \quad \Gamma, u{:}A_1; \Delta \vdash_{S,R} M : A_2}{\Gamma; \Delta \vdash_{S,R} \lambda u{:}A_1.M : \Pi u{:}A_1.A_2}$$

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

$$\frac{\Gamma; \Delta_1 \vdash_{S,R_1} M_1 : A_1 \multimap A_2 \quad \Gamma; \Delta_2 \vdash_{S,R_2} M_2 : A_1}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} M_1\,\hat{}\,M_2 : A_2} \qquad \frac{\begin{array}{c}\Gamma \vdash_S A_1 : P \quad \Gamma \vdash_S A_2 : P \\ \Gamma; \Delta \vdash_{S,R} M_1 : A_1 \quad \Gamma; \Delta \vdash_{S,R} M_2 : A_2\end{array}}{\Gamma; \Delta \vdash_{S,R} \langle M_1, M_2 \rangle : A_1 \mathbin{\&} A_2}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} M : A_1 \mathbin{\&} A_2}{\Gamma; \Delta \vdash_{S,R} \pi_i M : A_i}\ (i = 1, 2) \qquad \frac{}{\Gamma; \Delta \vdash_{S,R} \langle\rangle : \top} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M : A \quad \Gamma \vdash_S A = A' : P}{\Gamma; \Delta \vdash_{S,R} M : A'}$$

$$\frac{}{\Gamma; \epsilon \vdash_{S,\emptyset} \star : 1} \qquad \frac{\Gamma; \Delta_1 \vdash_{S,R_1} M_1 : A_1 \quad \Gamma; \Delta_2 \vdash_{S,R_2} M_2 : A_2}{\Gamma; \Delta_1, \Delta_2 \vdash_{S,R_1 \uplus R_2} \langle\!\langle M_1, M_2 \rangle\!\rangle : A_1 \otimes A_2}$$

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

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

$$\frac{}{\Gamma; \epsilon \vdash_{S,\{Mlk:A\}} Mlk = Mlk : A} \qquad \frac{\begin{array}{c}\Gamma, u{:}A \vdash_{S,R} M_1 = M_2 : B \\ \Gamma \vdash_S A_1 = A : P \quad \Gamma \vdash_S A_2 = A : P\end{array}}{\Gamma; \Delta \vdash_{S,R} \lambda u{:}A_1.M_1 = \lambda u{:}A_2.M_2 : \Pi u{:}A.B}$$

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

$$\frac{\begin{array}{c}\Gamma; \Delta_2 \vdash_{S,R_2} N_1 = N_2 : A \\ \Gamma; \Delta_1 \vdash_{S,R_1} M_1 = M_2 : A \multimap B\end{array}}{\Gamma; (\Delta_1, \Delta_2) \vdash_{S,R_1 \uplus R_2} M_1 \hat{}\, N_1 = M_2 \hat{}\, N_2 : B} \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A \quad \Gamma; \Delta \vdash_{S,R} N_1 = N_2 : B}{\Gamma; \Delta \vdash_{S,R} \langle M_1, N_1 \rangle = \langle M_2, N_2 \rangle : A \mathbin{\&} B}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A_1 \mathbin{\&} A_2}{\Gamma; \Delta \vdash_{S,R} \pi_i M_1 = \pi_i M_2 : A_i}\ (i = 1, 2) \qquad \frac{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A' \quad \Gamma \vdash_S A' = A : P}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A}$$

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

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

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

$$\frac{\Gamma \vdash_S A : P \quad \Gamma; \Delta, u\hat{:}B \vdash_{S,R} M_1 \hat{}\, u = M_2 \hat{}\, u : B}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A \multimap B} \qquad \frac{\begin{array}{c}\Gamma; \Delta \vdash_{S,R} \pi_1 M_1 = \pi_1 M_2 : A \\ \Gamma; \Delta \vdash_{S,R} \pi_2 M_1 = \pi_2 M_2 : B\end{array}}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : A \mathbin{\&} B}$$

$$\frac{\Gamma; \Delta \vdash_{S,R} M_1 : \top \quad \Gamma; \Delta \vdash_{S,R} M_2 : \top}{\Gamma; \Delta \vdash_{S,R} M_1 = M_2 : \top}$$

References

[1] Andrew W. Appel and Amy 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, January 2000.

[2] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Sirer, Marc Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, December 1995.

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

[4] Christopher Colby, Peter Lee, George Necula, and Fred 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] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, January 1999.

[6] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[8] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-99-159, Carnegie Mellon University, School of Computer Science, September 1999.

[9] 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.

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

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.

[13] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 2000. To appear. An earlier version appeared in the 1998 Workshop on Types in Compilation, volume 1473 of Lecture Notes in Computer Science.

[14] Greg Morrisett, David Walker, Karl Crary, and Neal 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.

[15] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementation*, pages 229–243, Seattle, October 1996.

[16] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, Berlin, Germany, March 2000.

[17] Joseph C. Vanderwaart and Karl Crary. A simplified account of the metatheory of linear LF. Technical report, Carnegie Mellon University, School of Computer Science, 2001.

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

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

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

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