A Cost-Effective Foundational Certified Code System

Susmit Sarkar

Abstract

Certified code systems enable untrusted programs to be proven safe to execute in a machine–checkable manner. Recent work has focused on building foundational certified code systems, where safety is defined relative to a concrete machine architecture. We wish to build a cost–effective system, with practicality along two dimensions — the intellectual effort to engineer the proofs, and the resource usage by the machine in verifying these proofs. Thus, we factor the proof that a particular program is safe to execute into two parts, a generic part and a program–specific part. These parts are linked by a mediating logic, typically a type system, which we call the safety condition. Consequently, we must prove that all programs that satisfy this condition are safe to execute, and then, we prove that the particular program satisfies this safety condition. Moreover, each of these proofs must be done in a cost–effective manner.

In previous work, we have described a machine–checkable proof for the first part, based on defining an operational semantics in LF and using the Twelf metalogic. For the second part, experience has shown that proof terms for a reasonable logic, or type system, are too big to generate, send across the network, and check. We wish to check adherence to the safety condition by an untrusted functional program. It remains to prove (in a machine–checkable manner) that the program implements the logic specified in a LF signature. We propose to accomplish this by static typechecking. We have designed an expressive type system using dependent refinements for this purpose.

1 Introduction

The widespread use of technology such as applets and browser plugins means that it is common for code to be written by untrusted or even unknown *code producers*, and downloaded and executed on trusted machines, the *code consumers*. The problem becomes even more acute in scenarios such as grid computing. In such systems, exemplified by SETI@Home [48], consumers download and run code to utilize spare cycles on their personal machines. To utilize the full potential of grid computing, we want to let arbitrary users program for the grid. Consumers are however understandably wary of this scenario. They are being asked to allow arbitrary programs, written by code producers they do not know or trust, execute on their personal machines. The ConCert project [6] at CMU seeks to build a system for trustless dissemination of software in the grid setting.

The technology we use to solve this problem is certified code. These systems, pioneered by Proof Carrying Code (PCC) [35], package a certificate proving the program is safe to execute together with the program. The certificate is a proof of safety of the program which can be checked in an automated manner. This check happens before the program gets executed at all. The code consumer now only has to check a proof of safety. The presumably harder burden of generating the proof is the responsibility of the code producer.

Early certified code systems [35, 34] prove safety by proving compliance with a type system. The knowledge of the type system is built into the system. These rely on the standard theorem of type safety – any program that type-checks is safe to execute. Such systems can be criticized on three counts.

- First, expressive type systems used in modern languages are complex. The type safety theorem then has a complex proof with many sub cases. The proofs of type safety are performed on paper, and require expert scrutiny to check. The process is tedious and error-prone.
- Second, the proof of safety is usually carried out relative to an abstract machine, at some distance from a real machine. The connection between the abstract machine and concrete architectures on which the code will actually run is not well-specified.

- Third, the type checker which actually checks compliance of code with the type system has to be trusted. There is no check to see whether it correctly implements the type system, or unsoundly accepts programs that should be disallowed.

More recent systems, called foundational [1, 22], seek to meet these concerns. They propose removing the type system from the list of components one must trust. Safety now is proved relative to a concrete machine architecture, such as the SPARC or the IA-32. Since a component of the trusted computing base is removed, this leads to greater confidence in the security of the system as a whole.

In a grid setting, foundational certified code systems have another compelling advantage. We want to enable using the grid for a large variety of participants, whose applications will vary substantially in their requirements. It is difficult to imagine that a single type system will be sufficient for all developers, since all type systems impose restrictions on the code they will accept. We want to have the flexibility of using different type systems, even ones that have not been invented yet.

Foundational certified code systems can give us the required flexibility. Code producers can, if they want, define their own type system, which they will use to write their code in. We no longer have to trust in this type system. Rather, the developer of the type system has to satisfy the consumer by proving meta-theoretic properties of his type system. The code consumer needs to have a machine-checkable proof of safety for this type system.

The goal of this thesis is then to build a cost–effective foundational certified code system. By cost–effective, we mean that we care about the efficient use of two resources. First, it must be feasible to ship the proofs and verify them with reasonable resources in computer time and space. The verification time is important because it is part of the consumer's overhead. Similarly, we would like to minimize the size of proofs or necessary annotations, as a factor of code size. Second, as a pragmatic matter, we would like to minimize the intellectual effort of proving safety, and engineering the proofs.

As a first step to being cost–effective in intellectual effort, we break the proof of safety into two natural parts. In common with similar systems, we isolate a whole class of programs by a logical system. We call this logic the *safety condition*. Then we produce two machine–checkable proofs. First, a generic (program–independent) proof says that any program in this class is safe to execute. Second, a program–specific proof says that the given program actually belongs in this class. We anticipate that most proofs will choose this mediating logic to be a type system, and the reader may wish to think of the safety condition as a type system. Then the first proof corresponds to a type safety proof, and the second proof corresponds to a type checking. Of course, both parts of the proof should be cost–effective as well.

Thus, in summary, my work leads me to make the following statement:

Thesis Statement

> A practical *certified code system*, with *machine-checkable* proofs of safety of code relative to a *concrete machine architecture*, can be built in a *cost-effective manner*, with proofs which are *small* enough to be packaged with code, *fast* to check, and with the proof infrastructure built with *reasonable intellectual effort.*

In the system described in this paper, we will use the TALT type system [11]. This is an expressive, low level type system. It is presented as an example of the kind of type system that will be used in the system. In accordance with our objectives, we will need to prove the safety of this type system. Further, we will need a provably correct way of checking adherence of a program to this type system. Both of these proofs will have to be machine-checkable.

The program–independent part of the proof deals with showing that any program that satisfies the safety condition is safe to execute on the machine. The proof techniques are well-known, based on defining an operational semantics as a logical system, and proving soundness between two logics. For our purposes, we want this proof to be machine–checkable. This part was done in previous work, and is discussed in Section 2.

The program–specific part of the proof deals with checking that a particular program satisfies the safety condition, an untrusted logic. This is the bulk of the remaining work of the thesis. We discuss the issues involved and our proposed solutions in Sections 3 and 4.

We will then discuss related work, and conclude by outlining the plan for the thesis.

2 Generic Proof Layer

The first part of our proof is a generic proof of safety. In the interest of factoring the effort of proof development, we isolate a whole class of programs via a formal logical system, which we call the *safety condition.* We then prove, in a machine-checkable manner, that any program within this class is safe to execute on the machine. To perform this proof, we need a formal specification of safety on the machine. We call this specification the *safety policy.* Notice that this specification must be trusted, and thus forms a part of the trusted components of the system. The safety condition, and the proof of safety need not be trusted.

We do have to decide on the representation language for our formalization. We choose to formalize in the Logical Framework (LF) [23], as implemented in the Twelf [42] system. This is well-suited for representing logics, and has been used to represent a wide variety of formal systems.

The ideas in this section were developed in our previous work [12]. We will give only a brief sketch in this section. For a more detailed account, take a look at that paper.

2.1 Safety Policy

Conceptually, the first task is to define the safety policy. In our work, we want to model concrete architectures. Our motivation being grid computing, a practical choice is the IA-32 [25] architecture, since that is the architecture used by the greatest number of participants on the grid. The particular machine chosen is not important, but we do have to choose some particular architecture.

The IA-32 is a complex architecture. We have implemented a subset of the architecture containing most of the commonly used instructions. We have formalized arithmetic and logical operations, as well as control transfer operations such as jumps, calls and returns. The formalization defines a notion of machine state, and the semantics of the instructions as a transition relation between machine states. In the usual parlance, this is an operational semantics for the IA-32 architecture.

The formalization is done so that the formal machine cannot perform any action that would be deemed unsafe. This means that if the actual machine is about to make a transition we deem unsafe, the formal machine will be stuck.

To simplify the statement of the safety policy, a special trap state is added to the formal machine. The machine can transition to this state only when the program stops execution in a safe manner. The trap state does not have any transitions to other states, but can always transition to itself.

The safety policy can then be stated as the formal machine should always be able to make a transition. The safety of all programs satisfying a particular safety condition is the content of a theorem we call the *safety theorem.* The proof of this theorem is filled in by the code producer, who defines a safety condition and proves the theorem for his safety condition.

Since the formal machine has no unsafe transitions, if it is not stuck, it must be making only safe transitions. We can argue that this implies safety on the concrete machine. The argument is easiest to understand in the contra-positive formulation. Suppose the concrete machine is going to make an unsafe transition. Then the formal machine, which follows the semantics step–by–step, must also be making the unsafe transition. Since there are no unsafe transitions in the formal machine, the formal machine must be stuck. The safety theorem then ensures that this does not happen.

2.2 Proof Infrastructure

We will now discuss our system and the proof infrastructure. As discussed above, a key trusted component is the safety policy, formalized within LF. The code consumer is responsible for specifying a logical condition, the safety condition, and proving the safety theorem for her choice of safety condition.

The safety theorem is a theorem in *metalogic*, *ie* a theorem about derivations in a logical system. It is certainly possible, if the logic is strong enough, to reason about these derivations in the same logic. Other efforts at foundational certified code have gone down this path, using higher order logic encoded in LF for Appel and Felty [2], and the calculus of inductive constructions for Hamid *et al* [22]. The cost to this approach is that there is a layer of encoding of metalogic into logic, which complicates reasoning.

Our system takes advantage of the Twelf metalogical framework [42]. Unlike the other systems mentioned, this is a logic designed explicitly for encoding and reasoning about other logics represented in LF. This design decision simplifies the engineering of proofs considerably. We have been able to develop our system with foundational safety proofs for an expressive type system [11] in less than two man-years.

Our generic proof is thus a proof in Twelf's metalogical system. As a practical matter, a code producer will want to reuse a safety condition for different programs. It would then be profitable to check the meta-proof once and then record the fact that the particular safety condition is known to be sound.

The cost to our approach is that proofs for the metalogic are more complicated to check. As a result, the proof checker for the Twelf metalogic is larger and more complicated than the checker for simpler logics. However, we believe the flexibility afforded by eliminating trusted components is a greater motivation for foundational certified code than increasing confidence by minimizing the size of the trusted computing base.

2.3 Proof of Safety Theorem

Our system defines its safety condition to be the TALT type system [11]. Recall that this is not a trusted type system, so the safety theorem has to be proved for TALT.

At a high level, this proof is the standard syntactic proof of type safety [52]. We define a type system, here TALT, and show that it is sound for the operational semantics of the IA-32 architecture. The proof uses familiar structural induction principles, and can be checked within the fragment of logic supported by Twelf's metalogical system.

However, in practice, there are two main complications. First, the IA-32 architecture is complex and has various idiosyncratic features. The details are abstracted away by defining a more idealized machine architecture to prove type safety results in. This abstract architecture is then proved sound relative to the concrete architecture.

Second, TALT itself is complex, being an expressive type system for low level code. It proved more elegant for the theoretical development [11] to let this be an implicitly typed language. This declarative formulation is not an operational definition, and is difficult to use to specify the behavior of a type checker. This is managed by defining a related type system XTALT (eXplicitly typed TALT), and proving it sound with respect to TALT.

Thus, the proof is structured in three steps:

1. **Static Stage:** a purely static portion that proves the soundness of the XTALT type system relative to the TALT type system.

2. **Abstract Stage:** a type safety proof for TALT relative to an abstract machine, using the usual lemmas of progress and type preservation. This was shown in Crary's previous work [11].

3. **Concrete Stage:** a type-free simulation argument that the TALT abstract machine is realized by the IA-32 architecture. This stage has to deal with the details of the concrete machine architecture. It transfers the results of the abstract type safety theorems to the concrete machine, and thus provides a proof of the safety theorem for well-typed XTALT programs.

3 Program-Specific Proof Layer

The program-specific part of the proof has to show, in a machine-checkable manner, that a particular program belongs to the safety condition. Recall that the safety condition is in practice a type system. Thus this step can be thought of as a typechecking problem. However, it differs in some important respects from the standard problem.

In a foundational system, the type system is not fixed *a priori*, and indeed, not trusted. This means that we cannot fix once and for all the type-checker. The solution has to be general enough to accommodate different type systems, and provide ways of verifying correctness in a machine-checkable manner.

In our proposed system, the type system used is TALT. This is an expressive type system, with many complexities, such as implicit typing and quantified types. In order to point out the key features of the problem and our solution, we will simplify the problem in this section. We will pretend that the untrusted type system is the explicitly typed simply typed lambda calculus, with a single base type `unitType`. The representation of the type system in Twelf's concrete syntax is given in Figures 1 and 2.

```
tp : type.

unitType : tp.
arrow    : tp -> tp -> tp.



exp : type.

unitTerm : exp.
app      : exp -> exp -> exp.
lam      : tp -> (exp -> exp) -> exp.
```

Figure 1: The simply typed lambda calculus : Syntax

```
of : exp -> tp -> type.

of_unit : of unitTerm unitType.
of_app  : of (app E1 E2) TP2
          <- of E1 (arrow TP1 TP2)
          <- of E2 TP1.
of_lam  : of (lam TP1 E) (arrow TP1 TP2)
          <- ( {x:exp}
                   of x TP1
                      -> of (E x) TP2).
```

Figure 2: The simply typed lambda calculus : Judgments

The typing judgement `of` takes a well-typed term `exp` and produces a derivation of its well-typedness at type `tp`. In accordance with the principles of higher-order abstract syntax, there is no rule for variables. Rather, when we add a term level variable to the context, we have to also add its typing assumption. Note that the term form for abstraction `lam` takes a term-level function. The corresponding type checking rule `of_lam` adds a new variable to the context, together with its typing.

3.1 Proposed Solution: Typing Derivations

The first, naive, idea is to send, and check, typing derivations. This solution is sufficiently general, since derivations can be represented in a logical framework such as LF independent of the particular type system. Checking that the program belongs to the type system is done by checking that the derivation is valid in the given type system. If derivations are represented in LF, the judgments–as–types and proofs–as–terms principle says that checking validity of derivations is done by checking well-typedness of the representing term in LF.

The problem is that derivations are typically very large in size. While small examples do not show the extent of the problem, requirements of space force us to give a small example. Consider the term

```
app (lam unitType ([x:tm] x)) (unit),
```

which represents the identity function at base type applied to unit in the type system presented above. The typing derivation for this term is

```
of_app of_unit (of_lam ([x:tm] [d:of x unitType] d)).
```

Here the typing derivation is of approximately the same size as the program. In more involved type systems, and in larger examples, the blowup has been noted to be of a factor of a thousand or more. Note that this is assuming we have type reconstruction of implicit arguments of LF available. The explicit form is larger still. Sending such large terms across the network, and typechecking them at the consumer side, is infeasible.

3.2 Proposed Solution : Oracle Based Checking

One possible refinement is to compress these derivations using the so-called "oracle" technique, due to Necula and Rahul [38]. Conceptually, checking that a proof is valid can be done as follows. The proof checker tries to prove the theorem. At any point where the proof tree branches, that is, a non-deterministic choice has to be made, the proof checker looks at the proof to see which branch to take. If the proof is valid, this procedure terminates in success without any backtracking. The information within the proof term can be represented by the sequence of choices made, and this sequence can be compactly encoded as a bit-string. Such bit-strings are called oracles in the literature [38].

In our system, since we do not trust the logic of the safety condition, we need to have a generic oracle-based proof system. On the code consumer side, this involves taking a proposition in a particular logic, and an oracle, and running the prover to verify that the oracle can produce a valid proof of the proposition. The prover consults the oracle whenever it faces a choice to determine which one to take. This procedure can verify theorems without any backtracking.

Dually, on the producer side, we have to produce this oracle string. This has to be done with knowledge of the verifying algorithm. The producer needs to know when a choice is necessary on the verifier side, and output the corresponding sequence of bits. Essentially, the same algorithm can be run on the producer, with the proof search taking as input a proof term instead of an oracle. When a choice is necessary, this procedure can consult the proof term to determine which branch to take, and output the necessary bits.

We implemented such an approach within the Twelf system [45]. The problem with this approach is a subtle one. In the system sketched out above, notice that the oracle producer takes in a proposition and a proof for the proposition, and converts that into an oracle. We have already discussed that the size of the derivation terms are very large. It turns out that they are unbearably large for even relatively modest sized programs. The terms do not fit in memory even for the initial typechecking and conversion at the producer side.

A possible solution would be to not have a generic oracle-producer. Instead, the assembler of the code generates the required oracles. This seems infeasible to engineer. The assembler has to have detailed knowledge of the verifying proof search algorithm to know when it will face a choice. This is an added level of complexity over the already complex task of assembling low-level code and verifying it is well-typed.

3.3 Proposed Solution : Typed Logic Programs

We abandon the effort to have typing derivation based approaches, and consider typechecking the code at the consumer side. If the type system was a trusted component, the type checker could also be a trusted component of the system. Unfortunately, in our system, the type system is untrusted and provided by the code producer. Then the type-checker also has to be provided by the producer. We need a guarantee that the type-checker correctly implements the purported type system. It should only accept programs belonging to the type system. Thus, we are looking at a way to provide a provably correct implementation.

One way of doing this is to use dependently typed logic programming, as illustrated by Appel and Felty [3]. We will illustrate this in the Twelf system. Twelf provides an operational interpretation of LF. Proof search can be done by systematically searching for terms inhabiting the type corresponding to the query. Then, any LF signature can be looked on as a logic program.

If proof search terminates in success, a proof term is produced. Assuming the proof search procedure is correct, this term has the type representing the proposition. Note that we do not have to run the program to know this fact. The type of the program is sufficient to ensure that if it terminates, the proposition belongs to the logic. This observation depends crucially on the fact that LF is strongly typed, and in fact, dependently typed.

The logical specification of the type system, such as in Figure 2 can thus be looked on as a certifying type-checker. We know that if it terminates in success, the program lies within the represented type system.

Types	$\tau ::=$	Unit	Unit Type
	$\mid$	$\tau_1 \to \tau_2$	Arrow Type
	$\mid$	$\tau_1 \times \tau_2$	Product Type
Terms	$e ::=$	unit	Unit
	$\mid$	$\mathsf{fun}\, x_1(x_2{:}\tau_1){:}\tau_2.e$	Functions
	$\mid$	$e_1\, e_2$	Applications
	$\mid$	$\langle e_1, e_2 \rangle$	Pairs
	$\mid$	$\pi_i\, e$	Projection from Pair
	$\mid$	error	Error

Figure 3: A Core Functional Language

In other words, partial correctness of the type-checker can be checked statically. This is particularly nice since the specification of the type system within LF has already been done in the generic part of the proof.

The problem is that the program we wrote in Twelf turns out to be very slow in practice. We want to use the type-checker on the consumer side, which make the time overhead an important factor. Also, this check has to be performed for every different piece of code before it is run, which means that amortizing the cost is not feasible.

When analysing the time performance of the program, the first problem is that arithmetic is slow. In a low-level type system, we have to do arithmetic while typechecking. In our system as described, we define the logical properties of natural numbers and binary bit-strings that we need. Then, all the arithmetic needed is formalized in terms of these definitions. Arithmetic using these is inherently slow, since we are working in unary, or for binary numbers, bit by bit. We would like to use the native capabilities of the machine to do arithmetic.

Twelf actually does possess constraint domains [43], which use the native arithmetic operations of the machine. We expect this to be faster than doing arithmetic in LF. However, the meta-theorem checker in Twelf does not work with constraint domains. We can check meta-theorems using the logical specification, and type-check using the constraint system. This leaves open the possibility that both versions do not implement the same system. This might still be all right, if we consider arithmetic to be part of the trusted computing base.

Actually, the real problem is that logic programming is a slow procedure. The only control mechanism available is depth-first search with backtracking. Further, since the logic programming engine is not tuned for a particular logic, unnecessary overhead can be imposed. For example, since the simply typed lambda calculus described above is syntax-directed and decidable, no backtracking information needs to be stored. A logic programming implementation will still store backtracking information at every point to cover possible failures.

4 Our Solution : Certifying Functional Type Checkers

What we really want to do is typechecking in a functional style, using a language such as SML [30]. The type checker should be written so that it can be proven partially correct, as above. However, it should still be possible to write a customized type checker, which can be tailored specifically for a particular type system.

We will develop the language to write such a typechecker incrementally in this section, as we discover requirements imposed by the task of writing a type checker that can be proven partially correct.

The core of our language is presented in Figure 3. This is a simple functional language, with recursive functions, unit as the base type, products and a single exception raising primitive error. The language is equipped with a conventional call-by-value semantics, which we elide in this presentation.

Sorts	$\gamma ::=$	$\Sigma \mathsf{u}{:}\gamma_1.\gamma_2$	Dependent Sums
	$\mid$	$*$	Unit Sort
	$\mid$	$\ldots$	
Indices	$\mathsf{i} ::=$	u	Variables
	$\mid$	$\langle \mathsf{i}_1, \mathsf{i}_2 \rangle$	Pairs
	$\mid$	$\pi_i \mathsf{i}$	Projection
	$\mid$	$\langle\rangle$	Unit Index
	$\mid$	$\ldots$	
Propositions	$P ::=$	$\mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma$	
	$\mid$	false	
Types	$\tau ::=$	$\ldots$	
	$\mid$	$\Pi \mathsf{u}{:}\gamma.\tau$	Universal Dependent Types
	$\mid$	$\Sigma \mathsf{u}{:}\gamma.\tau$	Existential Dependent Types
	$\mid$	$\mathsf{D}(\mathsf{i})$	Datatypes
Matches	$ms ::=$	$\cdot$	
	$\mid$	$\mathsf{C}\,[\mathsf{u}, x] \Rightarrow e \mid ms$	
Terms	$e ::=$	$\ldots$	
	$\mid$	$\mathsf{Fun}\, x(\mathsf{u}{:}\gamma){:}\tau.e$	
	$\mid$	$e\ [\mathsf{i}]$	
	$\mid$	$\mathsf{pack}\,\langle \mathsf{i}, e \rangle$	
	$\mid$	$\mathsf{let\, pack}\,\langle \mathsf{u}, x \rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$	
	$\mid$	$\mathsf{C}\,[\mathsf{i}, e]$	Datatype Constructors
	$\mid$	$\mathsf{case}^{\tau}\ e_1\ \mathsf{of}\ ms\ \mathsf{end}$	Case Expression
Signatures	$S ::=$	$\cdot$	
	$\mid$	$S, \mathsf{D}{:}\gamma \to \mathsf{TYPE}$	
	$\mid$	$S, \mathsf{C}{:}\,(\Pi \mathsf{u}{:}\gamma.\tau_1 \to \tau_2)$	
Contexts	$\Gamma ::=$	$\ldots$	
	$\mid$	$\Gamma, \mathsf{u}{:}\gamma$	
	$\mid$	Γ, P	

Figure 4: Additions for Indices and Sorts

4.1 Indexed Types

We have seen that we need dependent types to statically check partial correctness. However, full dependent types can lead to intractable typechecking problems. Fortunately, we do not need types dependent on terms. We follow Xi's approach in DML [57], by having types depend on objects from a simpler index domain.

We thus add a new syntactic class of indices, which are classified by sorts. These can be used in the types of the language by means of universal dependent types (Π types) and existential dependent types (Σ types) over indices. We propose an explicitly typed formalization for the index domain. This means that at the term level, we have constructors and destructors for these new constructs. Π types are introduced by functions taking index arguments, and eliminated by application to indices. Σ types are introduced by a `pack` construct, and eliminated by a `let pack` construct.

The typing rule for the new constructs is as follows:

$$\frac{\Gamma \vdash \gamma : \mathrm{sort} \quad \Gamma, \mathsf{u}{:}\gamma, x{:}(\Pi \mathsf{u}{:}\gamma.\tau) \vdash v : \tau}{\Gamma \vdash \mathsf{Fun}\, x(\mathsf{u}{:}\gamma){:}\tau.v : \Pi \mathsf{u}{:}\gamma.\tau} \qquad \frac{\Gamma \vdash e : \Pi \mathsf{u}{:}\gamma.\tau \quad \Gamma \vdash \mathsf{i} : \gamma}{\Gamma \vdash e\ [\mathsf{i}] : \tau\,[\mathsf{i}/\mathsf{u}]}$$

Notice that we have a value restriction on typing the bodies of a `Fun` expression. This is to avoid unsoundness in the presence of effects, as pointed out by Davies and Pfenning [14].

$$\frac{\Gamma, \mathsf{u}{:}\gamma \vdash \tau : \mathsf{TYPE} \quad \Gamma \vdash \mathsf{i} : \gamma \quad \Gamma \vdash e : \tau\,[\mathsf{i}/\mathsf{u}]}{\Gamma \vdash \mathsf{pack}\,\langle \mathsf{i}, e\rangle : \Sigma \mathsf{u}{:}\gamma.\tau} \qquad \frac{\Gamma \vdash e_1 : \Sigma \mathsf{u}{:}\gamma.\tau_1 \quad \Gamma, \mathsf{u}{:}\gamma, x{:}\tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\,\mathsf{pack}\,\langle \mathsf{u}, x\rangle = e_1\,\mathsf{in}\,e_2\,\mathsf{end} : \tau}$$

Finally, we have datatypes in the language. A type $\mathsf{D}(\mathsf{i})$ is a datatype indexed by a index i. Thus a datatype constructor D is a function from sorts to the kind of types. Constructors $\mathsf{C}\,[\mathsf{i}, e]$ construct members of the datatype. We assume each constructor takes one index term and a expression, and produces an inhabitant of a datatype. To allow multiple indices, we have pairs as well as unit in the index domain.

Our language assumes a set of datatypes and constructors already defined. These are given typing assumptions in a signature. We assume that the signature is well-formed, according to the rules below.

$$\frac{}{\vdash \cdot : \mathsf{ok}} \qquad \frac{\vdash S : \mathsf{ok} \quad \vdash_S \gamma : \mathsf{sort}}{\vdash S, \mathsf{D}{:}\gamma \rightarrow \mathsf{TYPE}} \qquad \frac{\begin{array}{c}\vdash S : \mathsf{ok} \quad \vdash_S \gamma : \mathsf{sort} \quad \mathsf{u}{:}\gamma \vdash_S \tau \\ \vdash_S \mathsf{D} : \gamma' \rightarrow \mathsf{TYPE} \quad \mathsf{u}{:}\gamma \vdash \mathsf{i} : \gamma'\end{array}}{\vdash S, \mathsf{C}{:}\,(\Pi \mathsf{u}{:}\gamma.\tau \rightarrow \mathsf{D}(\mathsf{i}))}$$

Datatypes are eliminated by a case construct. The case is annotated by the result type. The typing rule for this construct is given below.

$$\frac{\Gamma \vdash e_1 : \mathsf{D}(\mathsf{i}) \quad \Gamma \vdash \tau : \mathsf{ok} \quad \Gamma \vdash ms : \mathsf{D}(\mathsf{i}) \rightarrow \tau}{\Gamma \vdash \mathsf{case}^{\tau}\, e_1\, \mathsf{of}\, ms\, \mathsf{end} : \tau}$$

In the judgement form for checking matches $\Gamma \vdash ms : \mathsf{D}(\mathsf{i}) \rightarrow \tau$, all arguments must be thought of as supplied. Each branch of the case analyses a constructor. While type checking the body of the match, we are allowed to make additional assumptions regarding the particular branch we are in. This is spelled out in the rules for this judgement form.

$$\frac{S(\mathsf{D}) = \gamma \rightarrow \mathsf{TYPE} \quad \Gamma \vdash \mathsf{i} : \gamma \quad \Gamma \vdash \tau : \mathsf{TYPE}}{\Gamma \vdash \cdot : \mathsf{D}(\mathsf{i}) \rightarrow \tau} \qquad \frac{\begin{array}{c} S(\mathsf{C}) = \Pi \mathsf{u}{:}\gamma_1.\tau_1 \rightarrow \mathsf{D}(\mathsf{i}_2) \quad S(\mathsf{D}) = \gamma_2 \rightarrow \mathsf{TYPE} \\ \Gamma, \mathsf{u}{:}\gamma_1, x{:}\tau_1, \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma_2 \vdash e : \tau \quad \Gamma \vdash ms : \mathsf{D}(\mathsf{i}_1) \rightarrow \tau \end{array}}{\Gamma \vdash \mathsf{C}\,[\mathsf{u}, x] \Rightarrow e | ms : \mathsf{D}(\mathsf{i}_1) \rightarrow \tau}$$

While checking the body of the match, we introduce assumptions of an index variable and a term level variable of the appropriate types. More interestingly, we add an assumption that the return type of the constructor matches the type of the term being case analysed. This assumption takes the form of a proposition noting that the respective indices can be unified, since we constrain the datatype to be the same in the source of the case analysis and the head of the branch.

Thus, once we add different index domains, we will also need to reason about unification of indices from the domain. These will take the form of reasoning with the propositions. Notice that if unification can be shown to fail, this means that we made inconsistent assumptions. Looking at our rules, the match analysis rule is the only one that adds unification assumptions. If unification fails, this means that the corresponding branch is not reachable. Then, the branch is trivially well typed. This is the content of the following contradiction rule.

$$\frac{\Gamma \vdash \mathsf{false} \quad \Gamma \vdash \tau : \mathsf{TYPE}}{\Gamma \vdash e : \tau}$$

4.2 LF as an Index Domain

We have seen that a key requirement on our language is that it must be able to represent LF objects and families, since LF is our representation language for type systems. We will recapitulate the account of LF, which is very similar to Harper and Pfenning's account [24], with some changes we will point out.

LF is a dependently typed lambda calculus with objects classified by type families, which in turn are classified by kinds. There are dependent products at the family level, and dependent kinds. The system

LF Kinds	$\mathsf{K} ::=$	$\mathsf{type} \mid \Pi \mathsf{x{:}A.K}$
LF Families	$\mathsf{A} ::=$	$\mathsf{a} \mid \lambda \mathsf{x{:}A_1.A_2} \mid \mathsf{A\,M} \mid \Pi \mathsf{x{:}A_1.A_2}$
LF Objects	$\mathsf{M} ::=$	$\mathsf{c} \mid \mathsf{x} \mid \lambda \mathsf{x{:}A.M} \mid \mathsf{M_1\,M_2}$
LF Signatures	$\Sigma ::=$	$\cdot \mid \Sigma, \mathsf{a{:}K} \mid \Sigma, \mathsf{c{:}A}$
LF Contexts	$\Delta ::=$	$\cdot \mid \Delta, \mathsf{x{:}A}$

Figure 5: LF Syntax

$\vdash \Sigma : \mathsf{sig}$	Σ is a valid signature
$\Gamma \vdash \Delta : \mathsf{ctx}$	Δ is a valid context
$\Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}$	M has type A
$\Gamma; \Delta \vdash \mathsf{A} : \mathsf{K}$	A has kind K
$\Gamma; \Delta \vdash \mathsf{K} : \mathsf{kind}$	K is a valid kind
$\Gamma; \Delta \vdash \mathsf{M_1} = \mathsf{M_2} : \mathsf{A}$	$\mathsf{M_1}$ equals $\mathsf{M_2}$ at type A
$\Gamma; \Delta \vdash \mathsf{A_1} = \mathsf{A_2} : \mathsf{K}$	$\mathsf{A_1}$ equals $\mathsf{A_2}$ at kind K
$\Gamma; \Delta \vdash \mathsf{K_1} = \mathsf{K_2} : \mathsf{kind}$	$\mathsf{K_1}$ equals $\mathsf{K_2}$

Figure 6: Judgement Forms for LF

is sketched out in Figure 5. We also add family level abstractions, present in the original presentation of LF [23], but omitted from the Harper and Pfenning version of LF.

In LF, objects are defined relative to signatures, which declare the object-level and family-level constants in use. This means that the language is parametrized by the signature in use. We assume the signature is fixed to some particular signature in the following. In our application, this signature presumably contains the definition of the (untrusted) type system.

The judgement forms of the LF type system are given in Figure 6. We have judgements for well-formed terms, families and kinds. Our judgements take an extra argument, the core language level context. There are also judgements for definitional equality at each level. These rules axiomatize a congruence relation generated by $\beta\eta$-equality at both term and family levels.

Our first proposal is to have index domains of closed LF terms and LF families. Index equality on these domain is then definitional equality as defined above. Thus types of the core language can be refined by terms or families of the LF language. This proposal is summarized in Figure 7.

These are typed by the following rules:

$$\frac{\Gamma; \cdot \vdash \mathsf{A} : \mathsf{K}}{\Gamma \vdash [\mathsf{A}] : \ulcorner \mathsf{K} \urcorner} \qquad \frac{\Gamma; \cdot \vdash \mathsf{M} : \mathsf{A}}{\Gamma \vdash [\mathsf{M}] : \ulcorner \mathsf{A} \urcorner}$$

Notice that the terms and families used as indices are constrained to be closed, since the LF context used in typing them is empty.

In performing type checking in LF, we may need to use assumptions in the core level contexts. This would be the case, for example, if we need to use index variables. Thus, there are the constructs $[\![\mathsf{i}]\!]$ at both family and object levels. The typing rules for these constructs are:

$$\frac{\Gamma \vdash \mathsf{i} : \ulcorner \mathsf{K} \urcorner}{\Gamma; \Delta \vdash [\![\mathsf{i}]\!] : \mathsf{K}} \qquad \frac{\Gamma \vdash \mathsf{i} : \ulcorner \mathsf{A} \urcorner}{\Gamma; \Delta \vdash [\![\mathsf{i}]\!] : \mathsf{A}}$$

Notice that since index terms denote closed LF terms and families, we allow ourselves to explicitly weaken LF contexts by saying these objects are well-typed in *any* LF context.

$$
\begin{array}{lrl}
\text{Sorts} & \gamma ::= & \ldots \mid \ulcorner \mathsf{K} \urcorner \mid \ulcorner \mathsf{A} \urcorner \\
\text{Indices} & \mathsf{i} ::= & \ldots \mid [\mathsf{A}] \mid [\mathsf{M}] \\
\\
\text{LF Families} & \mathsf{A} ::= & \ldots \mid [\![\mathsf{i}]\!] \\
\text{LF Objects} & \mathsf{M} ::= & \ldots \mid [\![\mathsf{i}]\!]
\end{array}
$$

Figure 7: LF Terms and Families as Index Terms

Next, we need to give rules for reasoning with equality in these new index domains. At the index level, we lift definitional equality of LF families and objects.

$$\frac{\Gamma; \cdot \vdash \mathsf{A}_1 = \mathsf{A}_2 : \mathsf{K}}{\Gamma \vdash [\mathsf{A}_1] \doteq [\mathsf{A}_2] : \ulcorner \mathsf{K} \urcorner} \qquad \frac{\Gamma; \cdot \vdash \mathsf{M}_1 = \mathsf{M}_2 : \mathsf{A}}{\Gamma \vdash [\mathsf{M}_1] \doteq [\mathsf{M}_2] : \ulcorner \mathsf{A} \urcorner}$$

We also need to have equality in LF for the new constructs we have added.

$$\frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \ulcorner \mathsf{K} \urcorner}{\Gamma; \Delta \vdash [\![\mathsf{i}_1]\!] = [\![\mathsf{i}_2]\!] : \mathsf{K}} \qquad \frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \ulcorner \mathsf{A} \urcorner}{\Gamma; \Delta \vdash [\![\mathsf{i}_1]\!] = [\![\mathsf{i}_2]\!] : \mathsf{A}}$$

Next we need to give rules for reasoning about unification of these index constructs. Unfortunately, unification of our index terms, LF objects and families, involves higher-order unification. This is undecidable in general. Looking back at the way index constructs are used, we see that we need to reason about cases where unification will fail, not the cases where unification will succeed. Recall that index equality needs to be reasoned about only in the match rules for type checking cases of a case-analysis statement. It suffices to have a sound, but not necessarily complete axiomatization of unification. In the appendix, we present a set of inference rules for reasoning about inference. It is possible that this may need to be extended to a richer set.

4.3 Example: Simply Typed Lambda Calculus

To make our ideas concrete, we will develop a complete typechecker for the explicitly typed version of the simply typed lambda calculus within our language. This will point out further modifications and extensions we need to make to our language.

The first thing to do is to define datatypes for our terms and types. We start to develop these below. Some of these definitions will actually have to be changed in the final version. The constructors for the term datatype will be given with each case of the typechecker.

```
Tp          : ⌜tp⌝ -> TYPE
Context     : 1 -> TYPE
Exp         : ⌜exp⌝ -> TYPE

UnitType    : Π_:1. 1 -> Tp ⌜unitType⌝
Arrow       : Π⟨t1,t2⟩:⌜tp × tp⌝. Tp (t1) × Tp (t2) -> Tp (⌜arrow ⟦t1⟧ ⟦t2⟧⌝)
```

What we want of our typechecker is that it not only produces a type for the given expression, but also proves that there exists a derivation in LF of the typing.

Thus, the signature we want our typechecker to have is:

$$\mathsf{typecheck} : \Pi e{:}\ulcorner \mathsf{exp} \urcorner.\mathsf{Context}(\langle\rangle) \times \mathsf{Exp}(e) \rightarrow \Sigma t{:}\ulcorner \mathsf{tp} \urcorner.\Sigma \_{:}\ulcorner \mathsf{of}\ [\![e]\!]\ [\![t]\!] \urcorner.\mathsf{Tp}(t)$$

Notice that we do not care what the typing derivation is, as long as we know one exists.

4.3.1 Case: Base Type

The base type case is easy, and serves as an introduction to our methodology.

First, the definition of the `UnitTerm` construct.

```
UnitTerm : Exp([unitTerm]).
```

For comparison and intuition, let us recall how the typechecking case in a ordinary typechecker (written in ML-like syntax) will look like.

```
fun typecheck (_, UnitTerm) = UnitType
```

We will describe the version in our language, in a stylized ML-like syntax, which it is easy to see how to translate into our core language. Our version is

```
Fun typecheck (_) (_, UnitTerm _ _) = pack ([unitType] ,
                                            pack ([of_unit],
                                                  UnitType))
```

In this case, it is easy to come up with the LF `tp` argument, unitType, as well as the derivation of_unit.

4.3.2 Case: Application

We will consider the application case next. The constructor definition takes the two expressions, and constructs the new expression.

```
App : Π⟨e1,e2⟩:⌜exp⌝ × ⌜exp⌝. Exp(e1) × Exp(e2) -> Exp([app ⟦e1⟧ ⟦e2⟧]).
```

The typechecking case in an usual typechecker is

```
...
| typecheck (Ctx, App (E1, E2)) = let
                                     Tp12 = typecheck (Ctx, E1)
                                     Tp2 = typecheck (Ctx, E2)
                                  in
                                     case Tp12 of
                                           Arrow (Tp11, Tp12) => if (= Tp11 Tp2)
                                                                    then Tp12
                                                                    else error
                                         | UnitType  => error
                                  end
```

This case presents a problem for us. Notice that we have to check two types for equality, to prove that the domain type of the function matches the argument type. We need to analyse the structure of the two types to do this. However, we also need to unify the underlying LF forms of the types. To fix this problem, we need to make a small change to the LF `of` judgement. This change is detailed in Figure 8. The new rule `of_app'`, together with a new judgement form `eqtp`, can be proved sound with respect to the earlier system. In fact, this soundness proof can even be done in a machine-checkable form, within Twelf's metalogic.

With this change, we can now write the function in our setting. We will need an auxiliary function `checkEqTp` to check two types to be structurally equal, and produce a `eqtp` argument if they are. If not, it can just abort with an error. The definition of this function is presented in the appendix. In this section we will content ourselves with writing its type signature.

```
checkEqTp : Πt1:⌜tp⌝. Πt2:⌜tp⌝. Tp (t1) × Tp (t2) -> Σ_:⌜eqtp ⟦t1⟧ ⟦t2⟧⌝. Unit
```

With this function in our hand, we can now write our function.

```
eqtp  : tp -> tp -> type.

eqtp_unit        : eqtp unitType unitType.
eqtp_arrow       : eqtp (arrow TP11 TP12) (arrow TP21 TP22)
                   <- eqtp TP11 TP21
                   <- eqtp TP12 TP22.

of_app'          : of (app E1 E2) TP12
                   <- of E1 (arrow TP11 TP12)
                   <- of E2 TP2
                   <- eqtp TP11 TP2.
```

Figure 8: New LF rule for typing application form

```
...
| typecheck (_) (Ctx, App ⟨e1,e2⟩ (E1,E2)) =
               let
                   pack (tp1,d1,Tp1) = typecheck [e1] (Ctx, E1)
                   pack (tp2,d2,Tp2) = typecheck [e2] (Ctx, E2)
               in
                   case Tp1 of
                       Arrow ⟨tp11,tp12⟩ (Tp11,Tp22) =>
                            let
                               pack (d3, unit) = checkEqTp [tp11] [tp2] (Tp12,Tp2)
                            in
                               pack (tp22,
                                     pack ([of_app' ⟦d3⟧ ⟦d2⟧ ⟦d1⟧],
                                           Tp22))
                            end
                     | UnitType [_,_] => error
               end
```

4.4 Open Terms and LF contexts

So far we have dealt with closed terms. The two remaining cases are for checking abstractions and variables. Here we face a serious problem, since we can only deal with closed terms. The abstraction case, in an usual typechecker, looks like:

```
...
| typecheck (Ctx, Lam (TP1, E2)) = let
                                     Tp2 = typecheck (ContextCons (TP1, Ctx), E2)
                                    in
                                     Arrow (Tp1, Tp2)
                                    end
```

In the recursive call, we have to pass along a new context. It is not obvious what context to pass. A related, but more serious problem is that the expression E2 is an open term, which depends on an extended context. The problem here is that we want to manipulate the LF context, since in higher-order abstract syntax, object language variables are represented by LF variables.

The solution for this is to pass along representations of LF contexts. Thus, LF contexts should become part of the index domains. We add dependent sum (Σ) types and unit types to LF. These will abstract

$$\begin{array}{llll}
\text{LF Families} & \mathsf{A} ::= & \ldots & \\
 & \mid & 1 & \text{Unit Family} \\
 & \mid & \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 & \text{Dependent Sum Type} \\
\text{LF Objects} & \mathsf{M} ::= & \ldots & \\
 & \mid & \langle\rangle & \text{Unit Object} \\
 & \mid & \langle \mathsf{M}_1, \mathsf{M}_2 \rangle & \text{Pairs} \\
 & \mid & \pi_i \mathsf{M} & \text{Projection}
\end{array}$$

Figure 9: Σ and Unit Types in LF

over the LF context and be passed in cases where we need to work with contexts. The idea is presented in Schurmann's work [46], where a similar system called LF$^{\Sigma}$ is discussed. The changes to LF syntax are detailed in Figure 9.

The typing rules for the new constructs are given below.

$$\frac{}{\Gamma; \Delta \vdash 1 : \mathsf{type}} \qquad \frac{\Gamma; \Delta \vdash \mathsf{A}_1 : \mathsf{type} \quad \Gamma; \Delta, \mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Gamma; \Delta \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}}$$

We will need to change the definition of the `Context` datatype. The new datatype depends on a family, which is a representation of the LF context used.

$\texttt{Context : } \ulcorner \mathsf{type} \urcorner \to \mathsf{TYPE}$

The inhabitants of this datatype are the `Nil` and the `Cons` constructors. The `Cons` constructor takes the type argument it is adding to the context, and creates a new context formed by adding a new expression variable and its typing assumption. Thus it encodes only particular LF contexts, the ones following the "regular worlds" assumptions.

$\texttt{Nil : } \mathsf{Context}([1])$
$\texttt{Cons : } \Pi\langle \texttt{c,t} \rangle\texttt{:}\ulcorner \mathsf{type} \urcorner \times \ulcorner \mathsf{tp} \urcorner\texttt{. } \mathsf{Context}([\![\mathsf{c}]\!]) \times \mathsf{Tp}([\![\mathsf{t}]\!]) \texttt{ -> } \mathsf{Context}([[\![\mathsf{c}]\!] \times \Sigma \mathsf{e}{:}\mathsf{exp.of\ e}\ [\![\mathsf{t}]\!]])$

The `Exp` datatype also needs to change, since it will depend on LF expressions living in different LF contexts. We abstract over the LF context via a dependent sum type. Thus, `Exp` now depends on a representation of a LF context, and a function generating an `exp` in the context.

$\texttt{Exp : } \Sigma\texttt{c:}\ulcorner \mathsf{type} \urcorner\texttt{. } \ulcorner [\![\mathsf{c}]\!] \to \mathsf{exp} \urcorner \texttt{ -> } \mathsf{TYPE}$

The `Unit` and the `App` constructors also have to change, to reflect this new signature. Details are given in the appendix. We present the case for the `Lam` constructor below.

$\texttt{Lam : } \Pi\langle \texttt{c,t,e} \rangle\texttt{:(}\Sigma\texttt{c:}\ulcorner \mathsf{type} \urcorner\texttt{. } \ulcorner \mathsf{tp} \urcorner \times \ulcorner [\![\mathsf{c}]\!] \times \mathsf{exp} \to \mathsf{exp} \urcorner\texttt{)}$
$\quad \mathsf{Exp}(\langle [[\![\mathsf{c}]\!] \times \Sigma \mathsf{e}{:}\mathsf{exp.of\ e}\ [\![\mathsf{t}]\!]], [\lambda\gamma.[\![\mathsf{e}]\!]\ \langle \pi_1\gamma, \pi_{12}\gamma \rangle] \rangle) \times \mathsf{Tp}(\mathsf{t}) \texttt{ ->}$
$\qquad \mathsf{Exp}(\langle \mathsf{c}, \lambda\gamma : [\![\mathsf{c}]\!].\mathsf{lam}\ (\lambda \mathsf{x}{:}\mathsf{exp.e}\ \langle \gamma, \mathsf{x} \rangle) \rangle)$

This might be difficult to read, so we will explain in detail. The constructor takes three index argument. The first argument represents the context, the second the extending type, and the third one an expression living in the context given by the first argument extended by a `exp`. The constructor also takes two term level arguments. The first argument is a member of the `Exp` datatype. This represents a `exp` corresponding to that produced by the third index argument that lives in an extended context. The second term argument is a representation of the type argument given by the second index argument. The constructor produces a `Exp`, which lives in the context given by the first index argument. This represents an expression given by `lam` applied to the third index argument.

Finally, the signature of the `typecheck` function also changes. This now cannot return closed derivations, but has to return derivations living in contexts. Thus, it returns a function which takes an appropriate context and produces a derivation of the right type. The new signature is given below.

```
typecheck : Πc:⌜type⌝. Πe:⌜⟦c⟧ -> exp⌝.
                    Context [c] * Exp [c,e]
                             -> Σt:⌜tp⌝. Σd:⌜Πγ:⟦c⟧. of (⟦e⟧ γ) ⟦t⟧⌝. Tp [t]
```

The case for checking a `lam` expression can now be written.

```
...
| typecheck ctx [_] (Ctx, Lam ⟨_,tp1,e1⟩(TP1, E2)) =
         let
            pack (tp2,d1,Tp2) = typecheck [⌜⟦c⟧ × Σe1:exp. of e1 ⟦tp1⟧⌝]
                                                [⌜λγ. ⟦e⟧⟨π₁γ, π₁₂γ⟩⌝]
                                          ((Cons ⟨ctx,tp1⟩(Ctx,Tp1)), E2)
         in
            pack ( [arrow ⟦tp1⟧ ⟦tp2⟧],
                pack ( [λc:⟦ctx⟧.of_lam (λe2:exp.λd2:(of e2 tp1).⟦d1⟧ ⟨c, e2, d2⟩)],
                       Arrow ⟨tp1,tp2⟩ (Tp1, Tp2) ))
         end
```

Notice how closely this matches a standard type checker. The function producing derivations has to create a derivation using `of_lam` applied to an appropriate LF level function. This LF level function takes an expression and a typing assumption for the variable, and passes a packaged context to `d1`, the derivation function from the recursive call.

4.4.1 Case: Variable

The last remaining case is for typechecking variables. To check a variable, we have to search through the context for its typing assumption. A variable can only be a well-formed term if it is actually present in the context. This is easier to enforce if we have deBruijn representation of variables.

We create an auxiliary datatype `Index` to represent variables, where Index counts backwards from the end of the context.

```
Index : Σctx:⌜type⌝.⌜⟦ctx⟧ → exp⌝ → TYPE
```

The inhabitants of `Index` are the constructors `z` and `s` representing the natural number index.

```
Z : Π⟨ctx,t⟩:⌜type⌝ × ⌜tp⌝. 1 -> Index ([⟦ctx⟧ × Σe:exp.(of e ⟦t⟧)], [λγ : _. π₁₂γ])
Z : Π⟨ctx,t,e⟩:(Σctx:⌜type⌝. ⌜tp⌝ × ⌜⟦ctx⟧ → exp⌝).
        Index (ctx, m) -> Index ([⟦ctx⟧ × Σe:exp.(of e ⟦t⟧)], [λγ : _. e (π₁γ)])
```

Given this auxiliary datatype, the definition of the `Var` constructor is simple. It should take a `Index` object to indicate which variable it is, and return an `Exp` object living in the appropriate context.

```
Var : Π⟨c,e⟩:(Σc:⌜type⌝. ⌜⟦c⟧ → exp⌝). Index (c, e) -> Exp (c, e)
```

To type check the variable, we will break the work to a separate function, which we call `getTypeCtx`. This function analyzes the context, picking up the appropriate type by consulting the `Index` argument passed to it.

First, let us consider the case for the `Nil` context. Since no constructor for `Index` can construct a nil context, this case is unreachable by typing. We have to perform a case analysis on the `Index` argument to make this explicit for the type system.

```
getTypeCtx : Πc:⌜type⌝. Πe:⌜⟦c⟧ -> exp⌝.
                  Context (c) * Index (c, e)
                            -> Σt:⌜tp⌝. Σd:⌜Πγ:⟦c⟧. of (⟦e γ⟧ ⟦t⟧⌝. Tp (t)
Fun getTypeCtx ⟨c,e⟩ ((Nil _ _), i) = case i of Z _ _ => UNREACHABLE
                                              | S _ _ => UNREACHABLE
```

Now we consider the case when the context is non-empty. The base case is for `Index` to be `S`. In this case, the type can be picked up from the head of the context, and the top of the context also contains the typing derivation.

```
...
| getTypeCtx ⟨c,e⟩ ((Cons ⟨_,tp⟩ (_,Tp)), Z _) =
      pack (tp,
          pack ( [λc′:⟦c⟧.π22c′],
              Tp))
```

Finally, we have to treat the case where the `Index` is a successor, *ie* formed by a `S`. In this case, we have to make a recursive call, and package up the derivation appropriately.

```
...
| getTypeCtx ⟨c_orig,_⟩ (((Cons ⟨_,tp⟩ (c,Tp)), (S ⟨c1,t1,e1⟩ i)) =
      let
         pack (tp1,d1,Tp1) = getTypeCtx ⟨c1,e1⟩ (c, i)
      in
         pack (tp1 ,
             pack ( [λc′:⟦c_orig⟧.d1(π1  c′)],
                 Tp1))
      end
```

Finally, the main typechecker just defers the work to the `getTypeCtx` function just defined.

```
...
| typecheck ⟨c,e⟩ (context, Var _ ind) = getTypeCtx ⟨c,e⟩ (context, ind)
```

4.5 Recapitulation and Future Work

We defined a core functional language with types depending on index domains. The language itself is simple, with products, recursive functions and a single exception which cannot be handled. There is no polymorphism so far in the language.

Next, we specified an extension to LF with dependent product and unit types. We instantiated the index domains in the core language with closed LF terms and closed LF families. Index equality was defined as definitional equality.

Finally, we showed that with an appropriate definition of the datatypes, we can write a typechecker for the simply typed lambda calculus in this language. We need to show that this typechecker is a certifying type checker, in the sense that static type checking tells us that if it terminates in success, a valid LF derivation must exist.

The type system has been formalized using the Twelf system. We also have written the above type checker for the simply typed lambda calculus in the formalized system, and checked that it is well typed.

We conjecture the following propositions. Proof of these will form a theoretical underpinning of the proposed thesis.

Conjecture 4.1 *Type Checking of LF + Σ types + Unit type is decidable.*

Proof Sketch

This should be provable by following the methods of Harper and Pfenning [24]. We will have to come up with an algorithmic formulation of typing and prove it sound and complete with respect to the type system presented here.

Also note that this is a result claimed (without proof) for the language without unit type in Schurmann [46].

We have designed and proved correct an algorithm for deciding equality [44]. Current work in progress is in extending this to an algorithm for all typing judgments. □

Conjecture 4.2 *LF + Σ + Unit is conservative over LF.*

Proof Sketch

This is also a result claimed by Schurmann [46]. □

Conjecture 4.3 *Canonical Forms holds for LF + Σ + Unit.*

Proof Sketch

This is also a result claimed by Schurmann [46]. □

Conjecture 4.4 *Canonical Forms holds for our language.*
In particular, if $\cdot \vdash e : \Sigma \mathsf{u}{:}\gamma.\tau$*, and* e *is a value, then* $e = \mathsf{pack}\,\langle \mathsf{i}, e_1 \rangle$*, where* $\cdot \vdash \mathsf{i} : \gamma$ *and* e_1 *is a value with* $\cdot \vdash e_1 : \tau\,[\mathsf{i}/\mathsf{u}]$.
Also, if $\cdot \vdash \mathsf{i} : \ulcorner \mathsf{A} \urcorner$*, then* $\Gamma \vdash \mathsf{i} \doteq [\mathsf{M}] : \ulcorner \mathsf{A} \urcorner$ *with* $\cdot;\cdot \vdash \mathsf{M} : \mathsf{A}$.

Conjecture 4.5 *Subject reduction holds for our language.*
That is, if $\Gamma \vdash e_1 : \tau$ *and* $e_1 \mapsto e_2$*, then* $\Gamma \vdash e_2 : \tau$.

All this leads up to our main result, which we will use to justify our approach.

Proposition 4.6 *If* $\cdot \vdash e : \Sigma \mathsf{u}{:}\ulcorner \mathsf{A} \urcorner.\tau$ *and evaluation of* e *terminates in success (*i.e. $e \mapsto^* e_1$ *and* e_1 *is a value), then there exists an object* M *such that* $\vdash^{LF} \mathsf{M} : \mathsf{A}$ *and a term* e_2 *such that* $\cdot \vdash e_2 : \tau\,[[\mathsf{M}]/\mathsf{u}]$.

Proof

By subject reduction, $\cdot \vdash e_1 : \Sigma \mathsf{u}{:}\ulcorner \mathsf{A} \urcorner.\tau$.
By canonical forms, $e_1 = \mathsf{pack}\,\langle \mathsf{i}, e_2 \rangle$, with $\cdot \vdash \mathsf{i} : \ulcorner \mathsf{A} \urcorner$ and $\cdot \vdash e_2 : \tau\,[\mathsf{i}/\mathsf{u}]$.
By canonical forms in the index domain, $\mathsf{i} = [\mathsf{M}]$, with $\cdot;\cdot \vdash \mathsf{M} : \mathsf{A}$.
By canonical forms in LF+Σ+Unit, $\cdot;\cdot \vdash \mathsf{M} = \mathsf{M}_1 : \mathsf{A}$, with M_1 canonical of type A. By conservativity over LF, $\vdash^{LF} \mathsf{M}_1 : \mathsf{A}$. □

Recall that our typechecker has a signature

```
typecheck : Πc:⌜type⌝. Πe:⌜⟦c⟧ -> exp⌝.
                    Context [c] * Exp [c,e]
                              -> Σt:⌜tp⌝. Σd:⌜Πγ:⟦c⟧. of (⟦e⟧ γ) ⟦t⟧⌝. Tp [t]
```

For a closed term `prog` of type `exp`, and a representation `Prog`, we can apply the typechecker function to get

```
typecheck [[1], ⌜λ_:1. prog⌝] (Nil, Prog)
: Σt:⌜tp⌝. Σ_:⌜Πγ : 1.of prog ⟦t⟧⌝.Tp (t)
```

Assuming the checker terminates in success, and applying proposition 4.6 to the signature of our type checker says that there exists a `t` such that $\vdash^{LF}$`t : tp` and a e_1 such that

$\cdot \vdash e_1 : \Sigma\_{:}\ulcorner \Pi\gamma : 1.$`of prog t`$\urcorner.$`Tp (t)`.

Applying canonical forms again, we get a `d` such that $\cdot \vdash$`d : `$\ulcorner\Pi\gamma$` : 1. of prog t`$\urcorner$. Another application of canonical forms gives us a M such that $\cdot \vdash \mathsf{M} : \lambda\_{:}1.$`of prog t`. Canonical forms and substitution on LF gives us a M_1 such that $\cdot \vdash \mathsf{M}_1 :$ `(of prog t)`. Conservativity over LF gives us finally a derivation

$$\cdot \vdash^{LF} \mathsf{M}_1 : \texttt{(of prog t)}$$

Thus, if our type checker terminates in success, there must exist a typing derivation derivable in LF. Our type checker is thus a certifying typechecker.

One final point is that we noted that we need to axiomatize, in a sound but not necessarily complete manner, a unification procedure for LF terms. We have a preliminary axiomatization sketched out in the appendix. We conjecture that this is sound with respect to the algorithm sketched out in Pfenning [41]. Whether this set of rules is sufficient for writing more complex programs has to be seen from experiment.

5 Related Work

5.1 Certified Code

The use of certified code was pioneered by Proof Carrying Code (PCC), in the series of papers [36, 35, 39]. These describe an architecture where proofs of the safety of a code are packaged together with the code itself. The safety policy in this system is specified by the code consumer. A trusted *verification condition generator* (VCGen) looks at the code and produces a series of verification conditions, which are theorems that must be proved. The code producer has the responsibility of providing the proofs of these theorems.

In principle, PCC can encode arbitrary safety policies and provide proofs for such policies. In practice, however, the proofs have to be produced by the code producer in an automated manner. This is usually accomplished by starting with a typed high-level language, and maintaining high level invariants through the compilation process [37]. The compiler could thus emit the necessary proofs, for properties such as memory safety, or satisfying the bytecode verifier of the Java Virtual Machine Language. In the elaboration of this idea into the SpecialJ compiler for Java [9], the system encoded into the VCGen the knowledge that the source language was Java. In effect, the safety policy was tied to the type system of Java.

The TAL project defined a low-level type system for an idealized RISC-based assembly language [34, 33]. Type checking of the code provided assurance that the code was memory-safe, with the standard theorem of type safety. Type systems provided a formal logical specification of a class of programs guaranteed to be memory-safe. Further, a compiler could use typed intermediate languages to maintain type information throughout the compilation process. Thus, a typed high level language could be compiled to typed assembly language. These systems would fix the target type system, thus losing the flexibility of PCC in principle. However, the use of types could lead to a formal translation, and typing annotations should in principle be small.

The concrete realization of TAL was TALx86, a version of TAL specialized to the IA-32 architecture [32]. This specialized the idealized TAL to the IA-32 architecture. The metatheoretic statements of type safety was not proved, but was loosely justified by analogy to TAL.

5.2 Foundational Certified Code

Appel *et al.* [1, 2] advocated a more foundational approach, in which the type system is not trusted, but explicitly proved safe. This would rely only on the axioms of a trusted, well-understood logic (higher-order logic, in the FPCC approach). The safety of programs would be proved relative to a concrete architecture.

The FPCC system developed a denotational semantics of types in higher-order logic. Complex semantic arguments had to be encoded to deal with advanced type systems with references and recursive types. Next, a concrete architecture was modelled as a formal system. Safety of code was proved relative to this concrete architecture.

The proofs of program safety was divided into two parts, as in our work. Most of the work concentrates on the generic part of the proof, in which a machine-checkable proof of type safety is produced. Type systems designed with this objective in mind include that of Chen *et al.* [8]

Hamid *et al.* [22] developed a different proof for a foundational system. They mechanized a standard syntactic proof of type safety. This approach, based on defining an operational semantics for the type system, and proving safety by simple structural induction techniques, is considered to be simpler to generate in paper proofs [52]. Our work, like theirs, also uses the syntactic technique. Similar to our work, this work also split the proof obligation into two parts. They defined a type system called Featherweight TAL, and proved it safe relative to an actual machine architecture.

Foundational certified code systems improve the reliability of the system by removing a key trusted component, the type system. Minimizing the trusted computing base is a step towards increasing confidence in the system, since a smaller base can be more easily trusted. We may mention Bernard and Lee's *temporal logic PCC* [5], which removed the VCGen from a PCC system. This was done by specifying the safety policy in temporal logic, which led to both expressive safety policies as well as the elimination of a Verification Condition Generator. This simplification of the system by removing a big trusted component contributed to better confidence in the system.

5.3 Type Checking in Certified Code Systems

The problem of type checking in certified code systems has been less well studied. The work of Necula and Rahul [38] pioneered the oracle concept, which made possible extremely short encodings of proofs. This method is based on a view of guiding a theorem prover by describing the choices it must make to find a valid proof. If the choices that can be made by the theorem prover are predictable (that is, we know the theorem prover's algorithm), these choices can be encoded in an extremely compact manner. The system described in the work relied on a specialized logic, or type system.

For a system that does not have a trusted type system to do typechecking against, Wu *et al.* [53, 4] developed a logic program interpreter. This was a interpreter that could be written in a very small number of lines of code. The interpreter worked in a fragment of logic very close to Prolog. They observe that the logic program can operate as an effective type checker. For the reasons cited in this paper, we consider logic programming a half-way measure. Later, Appel and Felty observe [3] that dependent types can statically certify correctness of code. This avoids runtime errors in proof construction in the context of a theorem prover. measure. The key point of dependent types used to certify partial correctness is brought out well in this work.

5.4 Meta Programming

Our language is a functional language that manipulates terms from the LF type theory. This is related to the idea of meta-programming, in which programs create and manipulate other programs.

A system with very closely related aims is Schürmann *et al*'s Delphin project [16]. Within this project, the theory for a functional language manipulating LF objects is under development [47]. In this language, called the ∇-calculus, higher-order abstract syntax is utilized by using meta language variables to represent LF variables. This frees them from the need to represent or manage contexts. In order to avoid paradoxes, the language has a representation layer for LF objects, and a syntactically separated programming layer.

The operational model of the ∇-calculus is a hybrid of logic programming and functional programming. There is a non-deterministic pattern match operator, as also uses of existential (unification) variables. The treatment of variables differs significantly from ours. We explicitly represent LF level variables and contexts, programming their manipulation explicitly. In the ∇-calculus, there is no explicit manipulation, as these concepts are managed implicitly.

Metaprogramming systems have been studied from a variety of motivations, including partial evaluation and run-time code generation [51]. Studies in this area include Davies and Pfenning's series of papers [13, 15]. The MetaML programming language [50, 49] defines a functional language with additional primitives for meta programming. They describe a language with facilities for building and running code fragments. A type system has been defined for this language [31], taking ideas from Davies' work, which ensures that code fragments executed do not fail by trying to execute open code. The type system is proved safe, in the absence of effects or polymorphism. The major challenge in this line of work is to deal with open and closed code (*ie* code with free variables present or absent respectively) properly. Our work deals with the issue by closing code with respect to the context, in the manner that the Twelf system among others deal with contexts internally.

Pasalic *et al.* [40] build on the MetaML idea by defining a dependently typed language for staged computation. They show how to write a typechecker in their language. They use datatypes indexed by term level constructs. They reflect the term level in special constructs at the type constructor level, and use kinds to judge validity of such constructs.

5.5 Reflection in Type Theory

Our work is reminiscent of the use of reflection in type theories. This idea has been developed in the NuPRL project [10]. In such systems, the provability predicate is reflected into the language, and can be used in proofs like other predicates. This allows proofs of theorems to use metatheoretic arguments.

Since NuPRL is a programming language as well as a proof language, tactics to generate proofs can be programmed in it. Typing rules guarantee that the proofs produced are valid, a fact familiar from the LCF project [21]. The use of dependent types extends the notion of validity from merely being a valid proof of some theorem to the fact of being a valid proof of the particular theorem of interest. Finally, the use of reflection allows the writing of tactics that are provably correct.

Knoblock [27] discusses these methods in great detail. He divides tactics into three classes: complete tactics, partial tactics and search tactics. Complete tactics will produce a proof, while partial tactics will reduce the proof obligation. They are both guaranteed statically to succeed. Thus, if they apply, the tactic is guaranteed to terminate, and produce a solution or a reduced goal. A more general class of search tactics performs search, but if it terminates in success, is guaranteed to produce a valid proof. Our programs may be looked on as analogous in the sense that we prove partial correctness of programs.

Our proof representation language (LF) is separated from the programming language syntactically. NuPRL conflates these, to provide a powerful type theory. In NuPRL, we may even prove that a search tactic does not diverge, and thus satisfies the stronger property of total correctness. On the other hand, our system, being a weaker type system, enjoys decidable type checking,

5.6 Dependent Type Theories in Programming

Xi's DML system [57, 54] introduced dependent types in a ML-like language. This system had types dependent on terms from index domains. This work uses the index domain of natural numbers to point out the usefulness of the concept, such as in array bound check elimination [56].

This work was extended by Dunfield, in the series of papers [17, 18, 19]. These systems combine index domains such as Xi's with a powerful language of refinement types [20]. This seems to be more powerful than using just dependent types, through the use of property types and intersection and union types over such properties. We conjecture however that for properties in practical programs, a sufficiently powerful index domain can be defined to simulate the property language. This work also deals with type inference issues, by treating index domain constructs as implicitly typed. In contrast, our system is explicitly typed with the indices under programmer control.

Extending DML in a different direction, Xi *et al* [55] extended the notion of algebraic datatypes to inductive types at higher kinds, which they call guarded recursive datatypes. They showed the relation to DML style dependent types, and in a later paper [7], presented a direct translation from a dependently typed meta programming language into guarded recursive datatypes. Peyton Jones *et al* study type inference problems for similar systems [26]. These systems are related to our system in that we similarly extend the notion of datatypes to capture types dependent on index domain objects. Further, they have to solve similar problems in defining a type system for case analysis of types depending on index variables. Systems with guarded algebraic datatypes, or inductive type families, generalize pattern matching. Different typing assumptions can be made in checking different arms of a case statement. Similar to our system, unification has to be reasoned about, which the wobbly types work expresses by the notion of *type refinements*. We have not considered the problem of type inference so far. However, as explained above, we want programs in our system to be explicit about the indices they manage, so full type inference is not our goal in any case.

Similar systems of inductive datatypes are considered by McBride and McKinna [29]. They work in a very rich type theory [28], and add constructs and notation for allowing definitions of functions by pattern matching. Since their language allows only total functions, they can prove total correctness of programs. As an example, they prove the correctness of a type checker for the simply typed lambda calculus.

6 Conclusions and Road Map

The system we propose building meets the three objections we set out in the introduction. Since our proof of type-safety described in Section 2 is machine-checkable, we do not have to depend on expert human scrutiny. Further, the safety policy being specified in terms of concrete IA-32 architectures, the gap between abstract and concrete machines is removed. Finally, the certifying type-checkers described in Section 3 can be checked to be partially correct. This completes the objective of removing the type system from the list of trusted components in a safe yet efficient way.

I believe that my thesis will be accomplished by performing the following tasks:

1. Formulate a safety policy for the system by axiomatizing concrete details of the IA-32 architecture.

2. Write a generic safety proof for an example safety condition (TALT).
3. Formulate a type system to write a certifying type checker. This will have dependent indices over an extension of LF.
4. Prove meta theoretic properties of the type system.
5. Design a concrete syntax based on this system.
6. Write a type checker for this new language.
7. Write a compiler for the language. I envisage this to be a simple source translator to Standard ML, so that I can use existing compilers.
8. Write the certifying type checker (This is a different type checker, a checker for TALT written in the above language).
9. Write up the dissertation.

Of these, the first and second parts have been done in previous work. Minor extensions are proposed, to add more features to the language such as floating point support and a richer set of runtime features. The third part has formed the content of this proposal.

I anticipate the remaining parts to take approximately the amount of time listed.

Meta Theory	1 month
Design Concrete Syntax	1 month
Type Checker	4 months
Translator to SML	2 months
TALT Type Checker	3 months
Dissertation writing	4 months

In reality, I will work on many of these areas in parallel. However, assuming for the sake of argument that I tackle these tasks sequentially, this indicates I will need about 15 months from my proposal to complete my thesis.

References

[1] Andrew W. Appel. Foundational proof carrying code. In *Sixteenth IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] 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.

[3] Andrew W. Appel and Amy P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, 14(1):3–19, January 2004.

[4] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga. A trustworthy proof checker. *Journal of Automated Reasoning*, 31:231–260, 2003.

[5] Andrew Bernard and Peter Lee. Temporal logic for proof-carrying code. In *Eighteenth International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 31–46, Copenhagen, Denmark, July 2002. Springer-Verlag.

[6] Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless grid computing in ConCert. In *Third International Workshop on Grid Computing*, volume 2536 of *Lecture Notes in Computer Science*, pages 112–125, Baltimore, Maryland, November 2002.

[7] Chiyan Chen and Hongwei Xi. Meta-programming through typeful code representation. In *Eighth ACM International Conference on Functional Programming*, pages 275–286, Uppsala, Sweden, August 2003.

[8] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–219, June 2003.

[9] Christopher Colby, Peter Lee, George C Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[10] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[11] Karl Crary. Toward a foundational typed assembly language. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, Louisiana, January 2003.

[12] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. In *Nineteenth International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 106–120, Miami, Florida, August 2003. Springer-Verlag.

[13] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Eleventh IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[14] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ACM International Conference on Functional Programming*, September 2000.

[15] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 38(3):555–604, May 2001.

[16] Delphin. `http://cs-www.cs.yale.edu/homes/carsten/delphin/`, September 2004.

[17] Joshua Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, School of Computer Science, September 2002.

[18] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In A.D. Gordon, editor, *Sixth Foundations of Software Systems and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266, Warsaw, Poland, April 2003. Springer-Verlag.

[19] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In Xavier Leroy, editor, *Thirty-First ACM Symposium on Principles of Programming Languages*, pages 281–292, Venice, Italy, January 2004.

[20] Tim Freeman and Frank Pfenning. Refinement types for ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[21] Michael J C Gordon, Robin Milner, and C P Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[22] Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *Seventeenth IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

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

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

[25] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2001. Order numbers 245470–245472.

[26] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: Type inference for generalised algebraic data types. Draft, July 2004.

[27] Todd Knoblock. *Metamathematical Extensibility in Type Theory.* PhD thesis, Department of Computer Science, Cornell University, 1987.

[28] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.

[29] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004.

[30] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[31] Eugenio Moggi, Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. In S. D. Swierstra, editor, *Eighth European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

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

[33] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[34] 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.

[35] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[36] 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.

[37] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, June 1998.

[38] George Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 142–154, London, January 2001.

[39] George Ciprian Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, September 1998.

[40] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *ACM International Conference on Functional Programming*, Pittsburgh, Pennsylvania, January 2002.

[41] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Sixth IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, July 1991.

[42] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logic framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[43] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide, Version 1.3R4*, 2002. Available electronically at `http://www.cs.cmu.edu/~twelf`.

[44] Susmit Sarkar. The metatheory of an extension of LF with sum and unit types. In Preparation, March 2005.

[45] Susmit Sarkar, Brigitte Pientka, and Karl Crary. Small proof witnesses for the logical framework LF. Draft, January 2005.

[46] Carsten Schürmann. Towards functional programming with logical frameworks. Unpublished, available at http://cs-www.cs.yale.edu/homes/carsten/delphin/, July 2003.

[47] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus : Functional programming with higher-order encodings. In Preparation, September 2004.

[48] SETI@Home. `http://setiathome.ssl.berkeley.edu`, November 2000.

[49] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In *International Colloquium on Automata, Languages, and Programming*, pages 918–929, July 1998.

[50] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 203–217, June 1997.

[51] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), September 1998.

[52] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[53] Dinghao Wu, Andrew W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. In *Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 264–274, August 2003.

[54] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 1998.

[55] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Thirtieth ACM Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

[56] 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.

[57] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999.

A Abstract Syntax

We start with presenting the abstract syntax of LF$^{\Sigma,1}$. This is generated by the following grammar.

LF Kinds	K ::=	type	kind of types
	|	$\Pi\mathsf{x}{:}\mathsf{A}.\mathsf{K}$	dependent product kind
LF Families	A ::=	a	family constants
	|	$\lambda\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	family level abstraction
	|	$\mathsf{A}\,\mathsf{M}$	family application
	|	$\Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	family of functions
	|	$\Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2$	family of products
	|	$\mathsf{1}$	unit type
	|	$[\![\mathsf{i}]\!]$	injection from index
LF Objects	M ::=	c	object constants
	|	x	object variables
	|	$\lambda\mathsf{x}{:}\mathsf{A}.\mathsf{M}$	object functions
	|	$\mathsf{M}_1\,\mathsf{M}_2$	object level application
	|	$\langle\mathsf{M}_1,\mathsf{M}_2\rangle$	pairs of objects
	|	$\pi_i\mathsf{M}$	projections from pairs
	|	$\langle\rangle$	unit object
	|	$[\![\mathsf{i}]\!]$	injection from index
LF Signatures	Σ ::=	$\cdot$	empty
	|	$\Sigma,\mathsf{a}{:}\mathsf{K}$	extension by family level constant
	|	$\Sigma,\mathsf{c}{:}\mathsf{A}$	extension by object level constant
LF Contexts	Δ ::=	$\cdot$	empty
	|	$\Delta,\mathsf{x}{:}\mathsf{A}$	context extension

There are three levels of terms: objects, type families and kinds. Kinds classify families, and families belonging to type are called *types*. Types classify objects. The family level includes Π and Σ types, a unit type, as well as family level abstractions and applications.

We have object and family level constants. A signature records the types and kinds assigned to these constants. We only have object level variables, and these are provided types by contexts. Contexts can be assumed to be ordered lists. We assume that a variable is not bound more than once in contexts.

Next, we present the sorts and indices of our language.

Sorts	γ ::=	$\Sigma\mathsf{u}{:}\gamma_1.\gamma_2$	Dependent Sums
	|	$*$	Unit Sort
	|	$\ulcorner\mathsf{K}\urcorner$	Sort of LF kinds
	|	$\ulcorner\mathsf{A}\urcorner$	Sort of LF families
Indices	i ::=	u	Variables
	|	$\langle\mathsf{i}_1,\mathsf{i}_2\rangle$	Pairs
	|	$\pi_i\mathsf{i}$	Projection
	|	$\langle\rangle$	Unit Index
	|	$[\mathsf{A}]$	LF family index
	|	$[\mathsf{M}]$	LF object index
Propositions	P ::=	$\mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma$	Typed Index Equality
	|	false	Inconsistent Assumptions

Sorts classify indices. There is a base unit sort, dependent sums, and sorts for LF kinds and families. At the index level, correspondingly, there are unit, products, projections from products, and LF family and object indices. The LF object indices belong to corresponding LF family sort, and similarly LF family indices to LF kind sort.

Finally, we present the term and type levels of our language.

Types	$\tau ::=$	Unit	Unit Type
	$\mid$	$\tau_1 \to \tau_2$	Arrow Type
	$\mid$	$\tau_1 \times \tau_2$	Product Type
	$\mid$	$\Pi\mathsf{u}{:}\gamma.\tau$	Universal Dependent Types
	$\mid$	$\Sigma\mathsf{u}{:}\gamma.\tau$	Existential Dependent Types
	$\mid$	$\mathsf{D}(\mathsf{i})$	Datatypes
Matches	$ms ::=$	$\cdot$	
	$\mid$	$\mathsf{C}\,[\mathsf{u}, x] \Rightarrow e \mid ms$	
Terms	$e ::=$	unit	Unit
	$\mid$	$\mathsf{fun}\, x_1(x_2{:}\tau_1){:}\tau_2.e$	Functions
	$\mid$	$e_1\, e_2$	Applications
	$\mid$	$\langle e_1, e_2 \rangle$	Pairs
	$\mid$	$\pi_i\, e$	Projection from Pair
	$\mid$	error	Error
	$\mid$	$\mathsf{Fun}\, x(\mathsf{u}{:}\gamma){:}\tau.e$	Recursive functions taking index arguments
	$\mid$	$e\ [\mathsf{i}]$	Index Application
	$\mid$	$\mathsf{pack}\,\langle \mathsf{i}, e \rangle$	Package of Index and Expression
	$\mid$	$\mathsf{let\,pack}\,\langle \mathsf{u}, x \rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$	Unpacking a Pair
	$\mid$	$\mathsf{C}\,[\mathsf{i}, e]$	Datatype Constructors
	$\mid$	$\mathsf{case}^{\tau}\, e_1\ \mathsf{of}\ ms\ \mathsf{end}$	Case Expression
Signatures	$S ::=$	$\cdot$	
	$\mid$	$S, \mathsf{D}{:}\gamma \to \mathsf{TYPE}$	
	$\mid$	$S, \mathsf{C}{:}\,(\Pi\mathsf{u}{:}\gamma.\tau_1 \to \tau_2)$	
Contexts	$\Gamma ::=$	$\cdot$	
	$\mid$	$\Gamma, x{:}\tau$	
	$\mid$	$\Gamma, \mathsf{u}{:}\gamma$	
	$\mid$	Γ, P	

A subset of terms are judged values.

$$\begin{array}{lrl}
\text{Values} & v ::= & \mathsf{unit} \\
& \mid & \mathsf{fun}\, x_1(x_2{:}\tau_1){:}\tau_2.e \\
& \mid & \mathsf{Fun}\, x(\mathsf{u}{:}\gamma){:}\tau.e \\
& \mid & \mathsf{C}\,[\mathsf{i}, e] \\
& \mid & \langle v_1, v_2 \rangle \\
& \mid & \mathsf{pack}\,\langle \mathsf{i}, v \rangle
\end{array}$$

B Static Semantics

B.1 Judgement Forms

$\vdash \Sigma : \mathsf{sig}$	Σ is a valid signature
$\Gamma \vdash \Delta : \mathsf{ctx}$	Δ is a valid context
$\Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}$	M has type A
$\Gamma; \Delta \vdash \mathsf{A} : \mathsf{K}$	A has kind K
$\Gamma; \Delta \vdash \mathsf{K} : \mathsf{kind}$	K is a valid kind
$\Gamma; \Delta \vdash \mathsf{M}_1 = \mathsf{M}_2 : \mathsf{A}$	M_1 equals M_2 at type A
$\Gamma; \Delta \vdash \mathsf{A}_1 = \mathsf{A}_2 : \mathsf{K}$	A_1 equals A_2 at kind K
$\Gamma; \Delta \vdash \mathsf{K}_1 = \mathsf{K}_2 : \mathsf{kind}$	K_1 equals K_2
$\Gamma \vdash \gamma : \mathrm{sort}$	γ is a valid sort
$\Gamma \vdash \gamma_1 \equiv \gamma_2 : \mathrm{sort}$	γ_1 is equal to γ_2
$\Gamma \vdash \mathsf{i} : \gamma$	i is well formed at sort γ
$\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma$	i_1 is equal to i_2
$\vdash S$	S is a valid signature
$\Gamma \vdash \mathsf{ok}$	Γ is a valid context
$\Gamma \vdash \tau : \mathsf{TYPE}$	τ is a valid type
$\Gamma \vdash \tau_1 \equiv \tau_2 : \mathsf{TYPE}$	τ_1 and τ_2 are equal
$\Gamma \vdash ms : \mathsf{D}(\mathsf{i}) \to \tau$	ms is well-formed taking $\mathsf{D}(\mathsf{i})$ to τ
$\Gamma \vdash e : \tau$	e has type τ

B.2 Inference Rules

$\boxed{\vdash \Sigma : \mathsf{sig}}$

$$\frac{}{\vdash \cdot : \mathsf{sig}} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot \vdash_\Sigma \mathsf{K} : \mathsf{kind}}{\vdash \Sigma, \mathsf{a{:}K} : \mathsf{sig}} \qquad \frac{\vdash \Sigma : \mathsf{sig} \qquad \cdot \vdash_\Sigma \mathsf{A} : \mathsf{type}}{\vdash \Sigma, \mathsf{c{:}A} : \mathsf{sig}}$$

From now on we assume fixed a valid signature Σ and omit it from the judgments.

$\boxed{\Gamma \vdash \Delta : \mathsf{ctx}}$

$$\frac{}{\Gamma \vdash \cdot : \mathsf{ctx}} \qquad \frac{\Gamma \vdash \Delta : \mathsf{ctx} \qquad \Gamma; \Delta \vdash \mathsf{A} : \mathsf{type}}{\Gamma \vdash \Delta, \mathsf{x{:}A} : \mathsf{ctx}}$$

$\boxed{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}}$

$$\frac{\Delta(\mathsf{x}) = \mathsf{A}}{\Gamma; \Delta \vdash \mathsf{x} : \mathsf{A}} \qquad \frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Gamma; \Delta \vdash \mathsf{c} : \mathsf{A}} \qquad \frac{\Gamma; \Delta \vdash \mathsf{M}_1 : \Pi \mathsf{x{:}A}_2.\mathsf{A}_1 \qquad \Gamma; \Delta \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Gamma; \Delta \vdash \mathsf{M}_1\, \mathsf{M}_2 : \mathsf{A}_1\, [\mathsf{M}_2/\mathsf{x}]}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Gamma; \Delta, \mathsf{x{:}A}_1 \vdash \mathsf{M}_2 : \mathsf{A}_2}{\Gamma; \Delta \vdash \lambda \mathsf{x{:}A}_1.\mathsf{M}_2 : \Pi \mathsf{x{:}A}_1.\mathsf{A}_2} \qquad \frac{\Gamma; \Delta \vdash \mathsf{M}_1 : \mathsf{A}_1 \qquad \Gamma; \Delta \vdash \mathsf{M}_2 : \mathsf{A}_2\, [\mathsf{M}_1/\mathsf{x}]}{\Gamma; \Delta \vdash \langle \mathsf{M}_1, \mathsf{M}_2 \rangle : \Sigma \mathsf{x{:}A}_1.\mathsf{A}_2}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : \Sigma \mathsf{x{:}A}_1.\mathsf{A}_2}{\Gamma; \Delta \vdash \pi_1 \mathsf{M} : \mathsf{A}_1} \qquad \frac{\Gamma; \Delta \vdash \mathsf{M} : \Sigma \mathsf{x{:}A}_1.\mathsf{A}_2}{\Gamma; \Delta \vdash \pi_2 \mathsf{M} : \mathsf{A}_2\, [\pi_1 \mathsf{M}/\mathsf{x}]} \qquad \frac{}{\Gamma; \Delta \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash \mathsf{i} : \ulcorner \mathsf{A} \urcorner}{\Gamma; \Delta \vdash [\![\mathsf{i}]\!] : \mathsf{A}}$$

$$\frac{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}_1 \qquad \Gamma; \Delta \vdash \mathsf{A}_1 = \mathsf{A}_2 : \mathsf{type}}{\Gamma; \Delta \vdash \mathsf{M} : \mathsf{A}_2}$$

$\boxed{\Gamma;\Delta \vdash \mathsf{A} : \mathsf{K}}$

$$\frac{\Sigma(\mathsf{a}) = \mathsf{K}}{\Gamma;\Delta \vdash \mathsf{a} : \mathsf{K}} \qquad \frac{\Gamma;\Delta \vdash \mathsf{A}_1 : \mathsf{type} \quad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{K}}{\Gamma;\Delta \vdash \lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{K}} \qquad \frac{\Gamma;\Delta \vdash \mathsf{A}_1 : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{K} \qquad \Gamma;\Delta \vdash \mathsf{M} : \mathsf{A}_2}{\Gamma;\Delta \vdash \mathsf{A}_1\,\mathsf{M} : \mathsf{K}\,[\mathsf{M}/\mathsf{x}]}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Gamma;\Delta \vdash \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}} \qquad \frac{\Gamma;\Delta \vdash \mathsf{A}_1 : \mathsf{type} \qquad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{A}_2 : \mathsf{type}}{\Gamma;\Delta \vdash \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2 : \mathsf{type}}$$

$$\frac{}{\Gamma;\Delta \vdash 1 : \mathsf{type}} \qquad \frac{\Gamma \vdash \mathsf{i} : \ulcorner \mathsf{K} \urcorner}{\Gamma;\Delta \vdash [\![\mathsf{i}]\!] : \mathsf{K}} \qquad \frac{\Gamma;\Delta \vdash \mathsf{A} : \mathsf{K}_1 \qquad \Gamma;\Delta \vdash \mathsf{K}_1 = \mathsf{K}_2 : \mathsf{kind}}{\Gamma;\Delta \vdash \mathsf{A} : \mathsf{K}_2}$$

$\boxed{\Gamma;\Delta \vdash \mathsf{K} : \mathsf{kind}}$

$$\frac{}{\Gamma;\Delta \vdash \mathsf{type} : \mathsf{kind}} \qquad \frac{\Gamma;\Delta \vdash \mathsf{A} : \mathsf{type} \qquad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A} \vdash \mathsf{K} : \mathsf{kind}}{\Gamma;\Delta \vdash \Pi \mathsf{x}{:}\mathsf{A}.\mathsf{K} : \mathsf{kind}}$$

$\boxed{\Gamma;\Delta \vdash \mathsf{M}_1 = \mathsf{M}_2 : \mathsf{A}}$

$$\frac{\Delta(\mathsf{x}) = \mathsf{A}}{\Gamma;\Delta \vdash \mathsf{x} = \mathsf{x} : \mathsf{A}} \qquad \frac{\Sigma(\mathsf{c}) = \mathsf{A}}{\Gamma;\Delta \vdash \mathsf{c} = \mathsf{c} : \mathsf{A}} \qquad \frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \ulcorner \mathsf{A} \urcorner}{\Gamma;\Delta \vdash [\![\mathsf{i}_1]\!] = [\![\mathsf{i}_2]\!] : \mathsf{A}} \qquad \frac{\Gamma;\cdot \vdash \mathsf{M} : \mathsf{A}}{\Gamma;\cdot \vdash [\![[\mathsf{M}]]\!] = \mathsf{M} : \mathsf{A}}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{M}_{11} = \mathsf{M}_{21} : \Pi \mathsf{x}{:}\mathsf{A}_2.\mathsf{A}_1 \qquad \Gamma;\Delta \vdash \mathsf{M}_{12} = \mathsf{M}_{22} : \mathsf{A}_2}{\Gamma;\Delta \vdash \mathsf{M}_{11}\,\mathsf{M}_{12} = \mathsf{M}_{21}\,\mathsf{M}_{22} : \mathsf{A}_1\,[\mathsf{M}_{12}/\mathsf{x}]}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{A}_{11} = \mathsf{A}_1 : \mathsf{type} \qquad \Gamma;\Delta \vdash \mathsf{A}_{12} = \mathsf{A}_1 : \mathsf{type} \qquad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1 = \mathsf{M}_2 : \mathsf{A}_2}{\Gamma;\Delta \vdash \lambda \mathsf{x}{:}\mathsf{A}_{11}.\mathsf{M}_1 = \lambda \mathsf{x}{:}\mathsf{A}_{12}.\mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{A}_1 : \mathsf{type} \quad \{\Gamma;\Delta \vdash \mathsf{M}_1 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\} \quad \{\Gamma;\Delta \vdash \mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\} \quad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_1\,\mathsf{x} = \mathsf{M}_2\,\mathsf{x} : \mathsf{A}_2}{\Gamma;\Delta \vdash \mathsf{M}_1 = \mathsf{M}_2 : \Pi \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

$$\frac{\{\Gamma;\Delta \vdash \mathsf{A}_1 : \mathsf{type}\} \qquad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_{12} = \mathsf{M}_{22} : \mathsf{A}_2 \qquad \Gamma;\Delta \vdash \mathsf{M}_{11} = \mathsf{M}_{21} : \mathsf{A}_1}{\Gamma;\Delta \vdash (\lambda \mathsf{x}{:}\mathsf{A}_1.\mathsf{M}_{12})\,\mathsf{M}_{11} = \mathsf{M}_{22}\,[\mathsf{M}_{21}/\mathsf{x}] : \mathsf{A}_2\,[\mathsf{M}_{11}/\mathsf{x}]}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{M}_{11} = \mathsf{M}_{21} : \mathsf{A}_1 \qquad \Gamma;\Delta \vdash \mathsf{M}_{12} = \mathsf{M}_{22} : \mathsf{A}_2\,[\mathsf{M}_{11}/\mathsf{x}]}{\Gamma;\Delta \vdash \langle \mathsf{M}_{11}, \mathsf{M}_{12} \rangle = \langle \mathsf{M}_{21}, \mathsf{M}_{22} \rangle : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{M}_1 = \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Gamma;\Delta \vdash \pi_1 \mathsf{M}_1 = \pi_1 \mathsf{M}_2 : \mathsf{A}_1} \qquad \frac{\Gamma;\Delta \vdash \mathsf{M}_1 = \mathsf{M}_2 : \Sigma \mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2}{\Gamma;\Delta \vdash \pi_2 \mathsf{M}_1 = \pi_2 \mathsf{M}_2 : \mathsf{A}_2\,[\pi_1 \mathsf{M}_1/\mathsf{x}]} \qquad \frac{\Gamma;\Delta \vdash \mathsf{M}_1 : 1 \quad \Gamma;\Delta \vdash \mathsf{M}_2 : 1}{\Gamma;\Delta \vdash \mathsf{M}_1 = \mathsf{M}_2 : 1}$$

$$\frac{\Gamma;\Delta \vdash \mathsf{M}_1 = \mathsf{M}_3 : \mathsf{A}_1 \qquad \Gamma;\Delta \vdash \mathsf{M}_2 : \mathsf{A}}{\Gamma;\Delta \vdash \pi_1 \langle \mathsf{M}_1, \mathsf{M}_2 \rangle = \mathsf{M}_3 : \mathsf{A}_1} \qquad \frac{\Gamma;\Delta \vdash \mathsf{M}_1 : \mathsf{A}_1 \qquad \Gamma;\Delta,\mathsf{x}{:}\mathsf{A}_1 \vdash \mathsf{M}_2 = \mathsf{M}_3 : \mathsf{A}_2}{\Gamma;\Delta \vdash \pi_2 \langle \mathsf{M}_1, \mathsf{M}_2 \rangle = \mathsf{M}_3 : \mathsf{A}_2\,[\mathsf{M}_1/\mathsf{x}]}$$

$$\frac{\Gamma;\Delta\vdash \Sigma x{:}A_1.A_2 : \mathsf{type} \qquad \Gamma;\Delta\vdash \pi_1 M_1 = \pi_1 M_2 : A_1 \qquad \Gamma;\Delta\vdash \pi_2 M_1 = \pi_2 M_2 : A_2\,[\pi_1 M_1/x]}{\Gamma;\Delta\vdash M_1 = M_2 : \Sigma x{:}A_1.A_2}$$

$$\frac{\Gamma;\Delta\vdash M_2 = M_1 : A}{\Gamma;\Delta\vdash M_1 = M_2 : A} \qquad \frac{\Gamma;\Delta\vdash M_1 = M_2 : A \qquad \Gamma;\Delta\vdash M_2 = M_3 : A}{\Gamma;\Delta\vdash M_1 = M_3 : A}$$

$$\frac{\Gamma;\Delta\vdash A_1 = A_2 : \mathsf{type} \qquad \Gamma;\Delta\vdash M_1 = M_2 : A_2}{\Gamma;\Delta\vdash M_1 = M_2 : A_1}$$

$\boxed{\Gamma;\Delta\vdash A_1 = A_2 : K}$

$$\frac{\Sigma(a) = K}{\Gamma;\Delta\vdash a = a : K} \qquad \frac{\Gamma;\Delta\vdash A_{11} = A_1 : \mathsf{type} \quad \Gamma;\Delta\vdash A_{21} = A_1 : \mathsf{type} \quad \Gamma;\Delta, x{:}A_1\vdash A_{12} = A_{22} : K}{\Gamma;\Delta\vdash \lambda x{:}A_{11}.A_{12} = \lambda x{:}A_{21}.A_{22} : \Pi x{:}A_1.K}$$

$$\frac{\Gamma;\Delta\vdash A_1 = A_2 : \Pi x{:}A_3.K \qquad \Gamma;\Delta\vdash M_1 = M_2 : A_3}{\Gamma;\Delta\vdash A_1\,M_1 = A_2\,M_2 : K\,[M_1/x]}$$

$$\frac{\Gamma;\Delta\vdash A : \mathsf{type} \quad \{\Gamma;\Delta\vdash A_1 : \Pi x{:}A.K\} \quad \{\Gamma;\Delta\vdash A_2 : \Pi x{:}A.K\} \quad \Gamma;\Delta, x{:}A\vdash A_1\,x = A_2\,x : K}{\Gamma;\Delta\vdash A_1 = A_2 : \Pi x{:}A.K}$$

$$\frac{\{\Gamma;\Delta\vdash A : \mathsf{type}\} \qquad \Gamma;\Delta, x{:}A\vdash A_1 = A_2 : K \qquad \Gamma;\Delta\vdash M_1 = M_2 : A}{\Gamma;\Delta\vdash (\lambda x{:}A.A_1)\,M_1 = A_2\,[M_2/x] : K\,[M_1/x]}$$

$$\frac{\Gamma;\Delta\vdash A_{11} = A_{21} : \mathsf{type} \qquad \{\Gamma;\Delta\vdash A_{11} : \mathsf{type}\} \qquad \Gamma;\Delta, x{:}A_{11}\vdash A_{12} = A_{22} : \mathsf{type}}{\Gamma;\Delta\vdash \Pi x{:}A_{11}.A_{12} = \Pi x{:}A_{21}.A_{22} : \mathsf{type}}$$

$$\frac{\Gamma;\Delta\vdash A_{11} = A_{21} : \mathsf{type} \qquad \{\Gamma;\Delta\vdash A_{11} : \mathsf{type}\} \qquad \Gamma;\Delta, x{:}A_{11}\vdash A_{12} = A_{22} : \mathsf{type}}{\Gamma;\Delta\vdash \Sigma x{:}A_{11}.A_{12} = \Sigma x{:}A_{21}.A_{22} : \mathsf{type}}$$

$$\frac{}{\Gamma;\Delta\vdash 1 = 1 : \mathsf{type}} \qquad \frac{\Gamma\vdash i_1 \doteq i_2 : \ulcorner K\urcorner}{\Gamma;\Delta\vdash [\![i_1]\!] = [\![i_2]\!] : K} \qquad \frac{\Gamma;\cdot\vdash A : K}{\Gamma;\cdot\vdash [\![[A]]\!] = A : K} \qquad \frac{\Gamma;\Delta\vdash A_2 = A_1 : K}{\Gamma;\Delta\vdash A_1 = A_2 : K}$$

$$\frac{\Gamma;\Delta\vdash A_1 = A_2 : K \qquad \Gamma;\Delta\vdash A_2 = A_3 : K}{\Gamma;\Delta\vdash A_1 = A_3 : K} \qquad \frac{\Gamma;\Delta\vdash K_1 = K_2 : \mathsf{kind} \qquad \Gamma;\Delta\vdash A_1 = A_2 : K_2}{\Gamma;\Delta\vdash A_1 = A_2 : K_1}$$

$\boxed{\Gamma;\Delta\vdash K_1 = K_2 : \mathsf{kind}}$

$$\frac{}{\Gamma;\Delta\vdash \mathsf{type} = \mathsf{type} : \mathsf{kind}} \qquad \frac{\Gamma;\Delta\vdash A_1 = A_2 : \mathsf{type} \qquad \{\Gamma;\Delta\vdash A_1 : \mathsf{type}\} \qquad \Gamma;\Delta, x{:}A_1\vdash K_1 = K_2 : \mathsf{kind}}{\Gamma;\Delta\vdash \Pi x{:}A_1.K_1 = \Pi x{:}A_2.K_2 : \mathsf{kind}}$$

$$\frac{\Gamma;\Delta\vdash K_2 = K_1 : \mathsf{kind}}{\Gamma;\Delta\vdash K_1 = K_2 : \mathsf{kind}} \qquad \frac{\Gamma;\Delta\vdash K_1 = K_2 : \mathsf{kind} \qquad \Gamma;\Delta\vdash K_3 = K_3 : \mathsf{kind}}{\Gamma;\Delta\vdash K_1 = K_3 : \mathsf{kind}}$$

$\boxed{\Gamma \vdash \gamma : \textbf{sort}}$

$$\frac{}{\Gamma \vdash * : \mathrm{sort}} \qquad \frac{\Gamma \vdash \gamma_1 : \mathrm{sort} \quad \Gamma, \mathsf{u}{:}\gamma_1 \vdash \gamma_2 : \mathrm{sort}}{\Gamma \vdash \Sigma \mathsf{u}{:}\gamma_1 . \gamma_2 : \mathrm{sort}} \qquad \frac{\Gamma; \cdot \vdash \mathsf{K} : \mathsf{kind}}{\Gamma \vdash \ulcorner \mathsf{K} \urcorner : \mathrm{sort}} \qquad \frac{\Gamma; \cdot \vdash \mathsf{A} : \mathsf{type}}{\Gamma \vdash \ulcorner \mathsf{A} \urcorner : \mathrm{sort}}$$

$\boxed{\Gamma \vdash \gamma_1 \equiv \gamma_2 : \textbf{sort}}$

$$\frac{}{\Gamma \vdash * \equiv * : \mathrm{sort}} \qquad \frac{\Gamma \vdash \gamma_{11} \equiv \gamma_{21} : \mathrm{sort} \qquad \{\Gamma \vdash \gamma_{11} : \mathrm{sort}\} \qquad \Gamma, \mathsf{u}{:}\gamma_{11} \vdash \gamma_{12} \equiv \gamma_{22} : \mathrm{sort}}{\Gamma \vdash \Sigma \mathsf{u}{:}\gamma_{11}.\gamma_{12} \equiv \Sigma \mathsf{u}{:}\gamma_{21}.\gamma_{22} : \mathrm{sort}}$$

$$\frac{\Gamma; \cdot \vdash \mathsf{K}_1 = \mathsf{K}_2 : \mathsf{kind}}{\Gamma \vdash \ulcorner \mathsf{K}_1 \urcorner \equiv \ulcorner \mathsf{K}_2 \urcorner : \mathrm{sort}} \qquad \frac{\Gamma; \cdot \vdash \mathsf{A}_1 = \mathsf{A}_2 : \mathsf{type}}{\Gamma \vdash \ulcorner \mathsf{A}_1 \urcorner \equiv \ulcorner \mathsf{A}_2 \urcorner : \mathrm{sort}}$$

$$\frac{\Gamma \vdash \gamma_2 \equiv \gamma_1 : \mathrm{sort}}{\Gamma \vdash \gamma_1 \equiv \gamma_2 : \mathrm{sort}} \qquad \frac{\Gamma \vdash \gamma_1 \equiv \gamma_2 : \mathrm{sort} \quad \Gamma \vdash \gamma_2 \equiv \gamma_3 : \mathrm{sort}}{\Gamma \vdash \gamma_1 \equiv \gamma_3 : \mathrm{sort}}$$

$\boxed{\Gamma \vdash \mathsf{i} : \gamma}$

$$\frac{}{\Gamma \vdash \langle\rangle : *} \qquad \frac{\Gamma \vdash \mathsf{i}_1 : \gamma_1 \quad \Gamma \vdash \mathsf{i}_2 : \gamma_2\,[\mathsf{i}_1/\mathsf{u}]}{\Gamma \vdash \langle \mathsf{i}_1, \mathsf{i}_2 \rangle : \Sigma \mathsf{u}{:}\gamma_1.\gamma_2} \qquad \frac{\Gamma \vdash \mathsf{i} : \Sigma \mathsf{u}{:}\gamma_1.\gamma_2}{\Gamma \vdash \pi_1 \mathsf{i} : \gamma_1} \qquad \frac{\Gamma \vdash \mathsf{i} : \Sigma \mathsf{u}{:}\gamma_1.\gamma_2}{\Gamma \vdash \pi_2 \mathsf{i} : \gamma_2\,[\pi_1 \mathsf{i}/\mathsf{u}]}$$

$$\frac{\Gamma; \cdot \vdash \mathsf{A} : \mathsf{K}}{\Gamma \vdash [\mathsf{A}] : \ulcorner \mathsf{K} \urcorner} \qquad \frac{\Gamma; \cdot \vdash \mathsf{M} : \mathsf{A}}{\Gamma \vdash [\mathsf{M}] : \ulcorner \mathsf{A} \urcorner} \qquad \frac{\Gamma \vdash \mathsf{i} : \gamma_2 \quad \Gamma \vdash \gamma_1 \equiv \gamma_2 : \mathrm{sort}}{\Gamma \vdash \mathsf{i} : \gamma_1}$$

$\boxed{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma}$

$$\frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \Sigma \mathsf{u}{:}\gamma_1.\gamma_2}{\Gamma \vdash \pi_1 \mathsf{i}_1 \doteq \pi_1 \mathsf{i}_2 : \gamma_1} \qquad \frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \Sigma \mathsf{u}{:}\gamma_1.\gamma_2}{\Gamma \vdash \pi_2 \mathsf{i}_1 \doteq \pi_2 \mathsf{i}_2 : \gamma_2\,[\pi_1 \mathsf{i}_1/\mathsf{u}]} \qquad \frac{\Gamma \vdash \mathsf{i}_1 : * \quad \Gamma \vdash \mathsf{i}_2 : *}{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : *}$$

$$\frac{\Gamma \vdash \mathsf{i}_{11} \doteq \mathsf{i}_{21} : \gamma_1 \quad \Gamma \vdash \mathsf{i}_{12} \doteq \mathsf{i}_{22} : \gamma_2\,[\mathsf{i}_{11}/\mathsf{u}]}{\Gamma \vdash \langle \mathsf{i}_{11}, \mathsf{i}_{12} \rangle \doteq \langle \mathsf{i}_{21}, \mathsf{i}_{22} \rangle : \Sigma \mathsf{u}{:}\gamma_1.\gamma_2}$$

$$\frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_3 : \gamma_1 \qquad \Gamma \vdash \mathsf{i}_2 : \gamma}{\Gamma \vdash \pi_1 \langle \mathsf{i}_1, \mathsf{i}_2 \rangle \doteq \mathsf{i}_3 : \gamma_1} \qquad \frac{\Gamma \vdash \mathsf{i}_1 : \gamma_1 \qquad \Gamma, \mathsf{u}{:}\gamma_1 \vdash \mathsf{i}_2 \doteq \mathsf{i}_3 : \gamma_2}{\Gamma \vdash \pi_2 \langle \mathsf{i}_1, \mathsf{i}_2 \rangle \doteq \mathsf{i}_3 : \gamma_2\,[\mathsf{i}_1/\mathsf{u}]}$$

$$\frac{\Gamma \vdash \Sigma \mathsf{u}{:}\gamma_1.\gamma_2 : \mathrm{sort} \qquad \Gamma \vdash \pi_1 \mathsf{i}_1 \doteq \pi_1 \mathsf{i}_2 : \gamma_1 \qquad \Gamma \vdash \pi_2 \mathsf{i}_1 \doteq \pi_2 \mathsf{i}_2 : \gamma_2\,[\pi_1 \mathsf{i}_1/\mathsf{u}]}{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \Sigma \mathsf{u}{:}\gamma_1.\gamma_2}$$

$$\frac{\Gamma; \cdot \vdash \mathsf{A}_1 = \mathsf{A}_2 : \mathsf{K}}{\Gamma \vdash [\mathsf{A}_1] \doteq [\mathsf{A}_2] : \ulcorner \mathsf{K} \urcorner} \qquad \frac{\Gamma; \cdot \vdash \mathsf{M}_1 = \mathsf{M}_2 : \mathsf{A}}{\Gamma \vdash [\mathsf{M}_1] \doteq [\mathsf{M}_2] : \ulcorner \mathsf{A} \urcorner} \qquad \frac{\Gamma \vdash \mathsf{i} : \ulcorner \mathsf{K} \urcorner}{\Gamma \vdash [[\![\mathsf{i}]\!]] \doteq \mathsf{i} : \ulcorner \mathsf{K} \urcorner} \qquad \frac{\Gamma \vdash \mathsf{i} : \ulcorner \mathsf{A} \urcorner}{\Gamma \vdash [[\![\mathsf{i}]\!]] \doteq \mathsf{i} : \ulcorner \mathsf{A} \urcorner}$$

$$\frac{}{\Gamma, \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma}$$

$$\frac{\Gamma \vdash \mathsf{i}_2 \doteq \mathsf{i}_1 : \gamma}{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma} \qquad \frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma \quad \Gamma \vdash \mathsf{i}_2 \doteq \mathsf{i}_3 : \gamma}{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_3 : \gamma} \qquad \frac{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma_2 \quad \Gamma \vdash \gamma_1 \equiv \gamma_2 : \mathrm{sort}}{\Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma_1}$$

$\boxed{\vdash S}$

$$\frac{}{\vdash \cdot : \mathsf{ok}} \qquad \frac{\vdash S : \mathsf{ok} \quad \vdash_S \gamma : \mathsf{sort}}{\vdash S, \mathsf{D}{:}\gamma \to \mathsf{TYPE}} \qquad \frac{\begin{array}{c}\vdash S : \mathsf{ok} \quad \vdash_S \gamma : \mathsf{sort} \quad \mathsf{u}{:}\gamma \vdash_S \tau \\ \vdash_S \mathsf{D} : \gamma' \to \mathsf{TYPE} \quad \mathsf{u}{:}\gamma \vdash \mathsf{i} : \gamma'\end{array}}{\vdash S, \mathsf{C}{:}\, (\Pi\mathsf{u}{:}\gamma.\tau \to \mathsf{D}(\mathsf{i}))}$$

$\boxed{\Gamma \vdash \mathsf{ok}}$

$$\frac{}{\cdot \vdash \mathsf{ok}} \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma, x{:}\tau \vdash \mathsf{ok}} \qquad \frac{\Gamma \vdash \mathsf{ok} \quad \Gamma \vdash \gamma : \mathrm{sort}}{\Gamma, \mathsf{u}{:}\gamma \vdash \mathsf{ok}} \qquad \frac{\Gamma \vdash \mathsf{ok}}{\Gamma, \mathsf{false} \vdash \mathsf{ok}} \qquad \frac{\Gamma \vdash \mathsf{ok} \quad \Gamma \vdash \mathsf{i}_1 : \gamma \quad \Gamma \vdash \mathsf{i}_2 : \gamma}{\Gamma, \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma \vdash \mathsf{ok}}$$

$\boxed{\Gamma \vdash \tau : \mathsf{TYPE}}$

$$\frac{}{\Gamma \vdash \mathsf{Unit} : \mathsf{TYPE}} \qquad \frac{\Gamma \vdash \tau_1 : \mathsf{TYPE} \quad \Gamma \vdash \tau_2 : \mathsf{TYPE}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathsf{TYPE}}$$

$$\frac{\Gamma \vdash \tau_1 : \mathsf{TYPE} \quad \Gamma \vdash \tau_2 : \mathsf{TYPE}}{\Gamma \vdash \tau_1 \times \tau_2 : \mathsf{TYPE}} \qquad \frac{\Gamma \vdash \gamma : \mathrm{sort} \quad \Gamma, \mathsf{u}{:}\gamma \vdash \tau : \mathsf{TYPE}}{\Gamma \vdash \Pi\mathsf{u}{:}\gamma.\tau : \mathsf{TYPE}}$$

$$\frac{\Gamma \vdash \gamma : \mathrm{sort} \quad \Gamma, \mathsf{u}{:}\gamma \vdash \tau : \mathsf{TYPE}}{\Gamma \vdash \Sigma\mathsf{u}{:}\gamma.\tau : \mathsf{TYPE}} \qquad \frac{S(\mathsf{D}) = \gamma \to \mathsf{TYPE} \quad \Gamma \vdash \mathsf{i} : \gamma}{\Gamma \vdash \mathsf{D}(\mathsf{i}) : \mathsf{TYPE}}$$

$\boxed{\Gamma \vdash \tau_1 \equiv \tau_2 : \mathsf{TYPE}}$

$$\frac{}{\Gamma \vdash \mathsf{Unit} \equiv \mathsf{Unit} : \mathsf{TYPE}} \qquad \frac{\Gamma \vdash \tau_{11} \equiv \tau_{21} : \mathsf{TYPE} \quad \Gamma \vdash \tau_{12} \equiv \tau_{22} : \mathsf{TYPE}}{\Gamma \vdash \tau_{11} \to \tau_{12} \equiv \tau_{21} \to \tau_{22} : \mathsf{TYPE}}$$

$$\frac{\Gamma \vdash \tau_{11} \equiv \tau_{21} : \mathsf{TYPE} \quad \Gamma \vdash \tau_{12} \equiv \tau_{22} : \mathsf{TYPE}}{\Gamma \vdash \tau_{11} \times \tau_{12} \equiv \tau_{21} \times \tau_{22} : \mathsf{TYPE}} \qquad \frac{\Gamma \vdash \gamma_1 \equiv \gamma_2 : \mathrm{sort} \quad \Gamma, \mathsf{u}{:}\gamma_1 \vdash \tau_1 \equiv \tau_2 : \mathsf{TYPE}}{\Gamma \vdash \Pi\mathsf{u}{:}\gamma_1.\tau_1 \equiv \Pi\mathsf{u}{:}\gamma_2.\tau_2 : \mathsf{TYPE}}$$

$$\frac{\Gamma \vdash \gamma_1 \equiv \gamma_2 : \mathrm{sort} \quad \Gamma, \mathsf{u}{:}\gamma_1 \vdash \tau_1 \equiv \tau_2 : \mathsf{TYPE}}{\Gamma \vdash \Sigma\mathsf{u}{:}\gamma_1.\tau_1 \equiv \Sigma\mathsf{u}{:}\gamma_2.\tau_2 : \mathsf{TYPE}} \qquad \frac{S(\mathsf{D}) = \gamma \to \mathsf{TYPE} \quad \Gamma \vdash \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma}{\Gamma \vdash \mathsf{D}(\mathsf{i}_1) \equiv \mathsf{D}(\mathsf{i}_2) : \mathsf{TYPE}}$$

$$\frac{\Gamma \vdash \tau_2 \equiv \tau_1 : \mathsf{TYPE}}{\Gamma \vdash \tau_1 \equiv \tau_2 : \mathsf{TYPE}} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 : \mathsf{TYPE} \quad \Gamma \vdash \tau_2 \equiv \tau_3 : \mathsf{TYPE}}{\Gamma \vdash \tau_1 \equiv \tau_3 : \mathsf{TYPE}}$$

$\boxed{\Gamma \vdash ms : \mathsf{D}(\mathsf{i}) \to \tau}$

$$\frac{}{\Gamma \vdash \cdot : \mathsf{D}(\mathsf{i}) \to \tau} \qquad \frac{\begin{array}{c} S(\mathsf{C}) = \Pi\mathsf{u}{:}\gamma_1.\tau_1 \to \mathsf{D}(\mathsf{i}_2) \quad S(\mathsf{D}) = \gamma_2 \to \mathsf{TYPE} \\ \Gamma, \mathsf{u}{:}\gamma_1, x{:}\tau_1, \mathsf{i}_1 \doteq \mathsf{i}_2 : \gamma_2 \vdash e : \tau \quad \Gamma \vdash ms : \mathsf{D}(\mathsf{i}_1) \to \tau \end{array}}{\Gamma \vdash \mathsf{C}\,[\mathsf{u}, x] \Rightarrow e | ms : \mathsf{D}(\mathsf{i}_1) \to \tau}$$

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Unit}} \qquad \frac{}{\Gamma \vdash \mathsf{error} : \tau} \qquad \frac{\Gamma \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 \equiv \tau_2 : \mathsf{TYPE}}{\Gamma \vdash e : \tau_1} \qquad \frac{\Gamma \vdash \mathsf{false}}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma, x_1{:}\tau_1 \to \tau_2, x_2{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{fun}\, x_1(x_2{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, e : \tau_i}$$

$$\frac{\Gamma \vdash \gamma : \mathrm{sort} \quad \Gamma, \mathsf{u}{:}\gamma, x{:}(\Pi\mathsf{u}{:}\gamma.\tau) \vdash v : \tau}{\Gamma \vdash \mathsf{Fun}\, x(\mathsf{u}{:}\gamma){:}\tau.v : \Pi\mathsf{u}{:}\gamma.\tau} \qquad \frac{\Gamma \vdash e : \Pi\mathsf{u}{:}\gamma.\tau \quad \Gamma \vdash \mathsf{i} : \gamma}{\Gamma \vdash e\ [\mathsf{i}] : \tau\,[\mathsf{i}/\mathsf{u}]}$$

$$\frac{\Gamma \vdash \mathsf{i} : \gamma \quad \Gamma \vdash e : \tau\,[\mathsf{i}/\mathsf{u}]}{\Gamma \vdash \mathsf{pack}\, \langle \mathsf{i}, e \rangle : \Sigma\mathsf{u}{:}\gamma.\tau} \qquad \frac{\Gamma \vdash e_1 : \Sigma\mathsf{u}{:}\gamma.\tau_1 \quad \Gamma, \mathsf{u}{:}\gamma, x{:}\tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let\ pack}\, \langle \mathsf{u}, x \rangle = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : \tau}$$

$$\frac{S(\mathsf{C}) = \Pi\mathsf{u}{:}\gamma_1.\tau_1 \to \mathsf{D}(\mathsf{i}_2) \quad \Gamma \vdash \mathsf{i}_1 : \gamma_1 \quad \Gamma \vdash e : \tau_1\,[\mathsf{i}_1/\mathsf{u}]}{\Gamma \vdash \mathsf{C}\,[\mathsf{i}_1, e] : \mathsf{D}(\mathsf{i}_2\,[\mathsf{i}_1/\mathsf{u}])} \qquad \frac{\Gamma \vdash e_1 : \mathsf{D}(\mathsf{i}) \quad \Gamma \vdash \tau : \mathsf{ok} \quad \Gamma \vdash ms : \mathsf{D}(\mathsf{i}) \to \tau}{\Gamma \vdash \mathsf{case}^{\tau}\, e_1\ \mathsf{of}\ ms\ \mathsf{end} : \tau}$$

B.3 Unification Rules

$$\frac{\Gamma \vdash [1] \doteq [\Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2] : \ulcorner\mathsf{type}\urcorner}{\Gamma \vdash \mathsf{false}} \qquad \frac{\Gamma \vdash [\Sigma\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}] \doteq [\Sigma\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22}] : \ulcorner\mathsf{type}\urcorner}{\Gamma \vdash [\mathsf{A}_{11}] \doteq [\mathsf{A}_{21}] : \ulcorner\mathsf{type}\urcorner}$$

$$\frac{\Gamma \vdash [1] \doteq [\Pi\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2] : \ulcorner\mathsf{type}\urcorner}{\Gamma \vdash \mathsf{false}} \qquad \frac{\Gamma \vdash [\Pi\mathsf{x}{:}\mathsf{A}_{11}.\mathsf{A}_{12}] \doteq [\Pi\mathsf{x}{:}\mathsf{A}_{21}.\mathsf{A}_{22}] : \ulcorner\mathsf{type}\urcorner}{\Gamma \vdash [\mathsf{A}_{11}] \doteq [\mathsf{A}_{21}] : \ulcorner\mathsf{type}\urcorner}$$

$$\frac{\Gamma \vdash [\mathsf{a}_1] \doteq [\mathsf{a}_2] : \ulcorner\mathsf{K}\urcorner \quad \mathsf{a}_1 \neq \mathsf{a}_2}{\Gamma \vdash \mathsf{false}} \qquad \frac{\Gamma \vdash [\mathsf{a}_1\, \mathsf{M}_1] \doteq [\mathsf{a}_2\, \mathsf{M}_2] : \ulcorner\mathsf{K}\urcorner \quad \mathsf{a}_1 \neq \mathsf{a}_2}{\Gamma \vdash \mathsf{false}}$$

$$\frac{\Gamma \vdash [\mathsf{a}\, \mathsf{M}_1] \doteq [\mathsf{a}\, \mathsf{M}_2] : \ulcorner\mathsf{K}_1\urcorner \quad \Sigma(\mathsf{a}) = \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{K}}{\Gamma \vdash [\mathsf{M}_1] \doteq [\mathsf{M}_2] : \ulcorner\mathsf{A}\urcorner} \qquad \frac{\Gamma \vdash [\mathsf{c}_1] \doteq [\mathsf{c}_2] : \ulcorner\mathsf{A}\urcorner \quad \mathsf{c}_1 \neq \mathsf{c}_2}{\Gamma \vdash \mathsf{false}}$$

$$\frac{\Gamma \vdash [\mathsf{c}_1\, \mathsf{M}_1] \doteq [\mathsf{c}_2\, \mathsf{M}_2] : \ulcorner\mathsf{A}\urcorner \quad \mathsf{c}_1 \neq \mathsf{c}_2}{\Gamma \vdash \mathsf{false}} \qquad \frac{\Gamma \vdash [\mathsf{c}\, \mathsf{M}_1] \doteq [\mathsf{c}\, \mathsf{M}_2] : \ulcorner\mathsf{A}_1\urcorner \quad \Sigma(\mathsf{c}) = \Pi\mathsf{x}{:}\mathsf{A}.\mathsf{A}_2}{\Gamma \vdash [\mathsf{M}_1] \doteq [\mathsf{M}_2] : \ulcorner\mathsf{A}\urcorner}$$

$$\frac{\Gamma \vdash [\langle \mathsf{M}_{11}, \mathsf{M}_{12} \rangle] \doteq [\langle \mathsf{M}_{21}, \mathsf{M}_{22} \rangle] : \ulcorner\Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\urcorner}{\Gamma \vdash [\mathsf{M}_{11}] \doteq [\mathsf{M}_{21}] : \ulcorner\mathsf{A}_1\urcorner} \qquad \frac{\Gamma \vdash [\langle \mathsf{M}_{11}, \mathsf{M}_{12} \rangle] \doteq [\langle \mathsf{M}_{21}, \mathsf{M}_{22} \rangle] : \ulcorner\Sigma\mathsf{x}{:}\mathsf{A}_1.\mathsf{A}_2\urcorner}{\Gamma \vdash [\mathsf{M}_{12}] \doteq [\mathsf{M}_{22}] : \ulcorner\mathsf{A}_1\,[\mathsf{M}_{11}/\mathsf{x}]\urcorner}$$

C Type Checker for Simply Typed Lambda Calculus

C.1 LF definitions

The `tp` family represents the types of the calculus, with `unitType` being the base type and `arrow` being the arrow constructor.

```
tp          : type.

unitType    : tp.
arrow       : tp -> tp -> tp.
```

The `exp` family represents the terms of the calculus, with `unitTerm` representing the term constant, `lam` representing the lambda abstraction (with a rigid type), and `app` representing the application.

```
exp         : type.

unitTerm    : exp.
lam         : tp -> (exp -> exp) -> exp.
app         : exp -> exp -> exp.
```

Now we present the judgements of the calculus. The judgement `eqtp` represents structural equality between two types, and `of` represents typing a `exp` at a type `tp`. Notice that we do not have any implicit types in this formalism, all parameters are explicit. We still use Twelf's concrete syntax, so `{x:A} _`represents the dependent product constructor $\Pi x{:}A.\_$, and a `->` represents a non-dependent function type.

```
eqtp        : tp -> tp -> type.

eqtp_unit   : eqtp unitType unitType.
eqtp_arrow  : {tp11:tp} {tp12:tp} {tp21:tp} {tp22:tp}
                eqtp tp11 tp12
                -> eqtp tp12 tp22
                -> eqtp (arrow tp11 tp12) (arrow tp21 tp22).

of          : exp -> tp -> type.

of_unit     : of unitTerm unitType.
of_app      : {tp1:tp} {tp1':tp} {tp2:tp} {e1:exp} {e2:exp}
                eqtp tp1 tp1'
                -> of e2 tp1'
                -> of e1 (arrow tp1 tp2)
                -> of (app e1 e2) tp2.
of_lam      : {tp1:tp} {tp2:tp} {e:exp -> exp}
                (x:exp of x tp1
                      -> of (e x) tp2)
                -> of (lam tp1 e) (arrow tp1 tp2).
```

C.2 Datatype and Constructor definitions

Now we present datatypes and constructor definitions for the core language. The `Tp` datatype represents the types of the calculus. Some inferrable annotations to λ forms in LF are replaced by $\bullet$.

```
Tp          : ⌜tp⌝ -> TYPE

UnitType    : Π_:1. Unit -> Tp (⌜unitType⌝)
Arrow       : Π⟨tp1,tp2⟩:(⌜tp⌝ × ⌜tp⌝). Tp (tp1) -> Tp (tp2)
                -> Tp (⌜arrow ⟦tp1⟧ ⟦tp2⟧⌝)
```

The `Context` datatype represents the context used for typechecking. It is indexed by a representation of the context terms (and derivations) live in.

```
Context     : ⌜type⌝ -> TYPE

Nil         : Π_:1. Unit -> Context [1]
Cons        : Π⟨c,t⟩:(⌜type⌝ × ⌜tp⌝). Context (c) -> Tp (t)
                  -> Context (⌜⟦c⟧ × Σe:exp.of e ⟦t⟧⌝)
```

The `Index` datatype indexes into the current context to pick out a term level variable from the context.

```
Index       : (Σc:⌜type⌝. ⌜⟦c⟧ -> exp⌝) -> TYPE

Z           : Π⟨c,t⟩:⌜type⌝ × ⌜tp⌝. Unit ->
                   Index (⌜⟦c⟧ × Σe:exp.of e ⟦t⟧⌝ , ⌜λγ:•.$\pi_{12}\gamma$⌝)
S           : Π⟨c,t,e⟩:(Σc:⌜type⌝. ⌜tp⌝ × ⌜⟦c⟧ -> exp⌝). Index (c,e)
                    -> Index (⌜⟦c⟧ × Σe:exp.of e ⟦t⟧⌝ , ⌜λγ:•.e ($\pi_1\gamma$)⌝)
```

Finally, the `Exp` datatype represents terms of the calculus. It is indexed by a context and a LF function which produces a `exp` in the current context.

```
Exp         : (Σc:⌜type⌝. ⌜⟦c⟧ -> exp⌝) -> TYPE

Var         : Π⟨c,e⟩:(Σc:⌜type⌝. ⌜⟦c⟧ -> exp⌝). Index (c, e) -> Exp (c, e)
UnitTerm    : Πc:⌜type⌝. Unit -> Exp (c, ⌜λ_:⟦c⟧. unitTerm⌝)
App         : Π⟨c,e1,e2⟩:(Σc:⌜type⌝. ⌜⟦c⟧ -> exp⌝ × ⌜⟦c⟧ -> exp⌝).
                   Exp (c, e1) * Exp (c, e2)
                           -> Exp (c, ⌜λγ:⟦c⟧.app (⟦e1⟧ γ) (⟦e2⟧ γ)⌝)
Lam         : Π⟨c,t,efunc:(Πc:⌜type⌝. ⌜tp⌝ × ⌜⟦c⟧ × exp -> exp⌝.
                 Exp (⌜⟦c⟧ × Σe:exp.of e ⟦t⟧⌝ , ⌜λγ:•.efunc ⟨$\pi_1\gamma, \pi_{12}\gamma$⟩⌝) * Tp (t)
                   -> Exp (c, ⌜λγ:⟦c⟧.lam ⟦t⟧ (λe:exp.efunc ⟨γ, e⟩)⌝)
```

C.3 Auxiliary Functions

The `getTypeCtx` function takes a context and a index into that context, and returns a type with typing derivation of the corresponding variable.

```
getTypeCtx : Πc:⌜type⌝. Πe:⌜⟦c⟧ -> exp⌝.
                   Context (c) * Index (c,e)
                            -> Σt:⌜tp⌝. Σd:⌜Πγ:⟦c⟧. of (⟦e⟧ γ) ⟦t⟧⌝. Tp [t]

Fun getTypeCtx [_] [_] (Nil , Z _) = UNREACHABLE
 |  getTypeCtx [_] [_] (Nil , (S _ _)) = UNREACHABLE
 |  getTypeCtx [_] [_] ((Cons ⟨c, t⟩ (context,tp)), (Z _)) =
          pack (t,
                pack (⌜λγ:(⟦c⟧ × Σe:exp.of e ⟦t1⟧). $\pi_{22}\gamma$⌝,
                      tp))
 |  getTypeCtx [_] [_] ((Cons ⟨c, t⟩ (context, tp)) , (S ⟨c1, t1, e⟩ index)) =
          let
             pack (t_out, d_out, tp_out) = getTypeCtx ⟨c,e⟩ (context, index)
          in
                pack (t_out,
                      pack (⌜λγ:(⟦c⟧ × Σe:exp.of e ⟦t1⟧).⟦d_out⟧($\pi_1\gamma$)⌝,
                            tp_out))
          end
```

The `checkEqTp` performs the structural equality test on two types passed to it. If the equality test fails, it throws an error.

```
checkEqTp : Πtp1:⌜tp⌝. Πtp2:⌜tp⌝.
                Tp (tp1) * Tp (tp2) -> Σd:⌜eqtp ⟦tp1⟧ ⟦tp2⟧⌝. Unit

Fun checkEqTp [tp1] [tp2] ((UnitType _ _), (UnitType _ _)) =
     pack ([eqtp_unit] , unit)
 |  checkEqTp [tp1] [tp2] ((UnitType _ _), (Arrow _ _)) =
     error
 |  checkEqTp [tp1] [tp2] ((Arrow _ _), (UnitType _ _)) =
     error
 |  checkEqTp [tp1] [tp2] ((Arrow ⟨tp11, tp12⟩ (Tp11, Tp12)),
                               (Arrow ⟨tp21, tp22⟩ (Tp21, Tp22))) =
         let
               pack (d1, _) = checkEqTp [tp11, tp21] (Tp11, Tp21)
               pack (d2, _) = checkEqTp [tp12, tp22] (Tp12, Tp22)
         in
               pack (⌜eqtp_arrow ⟦t11⟧ ⟦t12⟧ ⟦t21⟧ ⟦t22⟧ ⟦d1⟧ ⟦d2⟧⌝, unit)
      end
```

C.4 Main Type Checking Function

```
typecheck : Πc:⌜type⌝. Πe:⌜⟦c⟧ -> exp⌝.
                 Context (c) * Exp (c,e)
                       -> Σt:⌜tp⌝. Σd:⌜Πγ:⟦c⟧. of (⟦e⟧ γ) ⟦t⟧⌝. Tp [t]

Fun typecheck [c] [e] (context, (Var _ ind)) = getTypeCtx [c] [e] (context, ind)
 |  typecheck [c] [_] (_, (UnitTerm _ _)) =
        pack ([unitType] ,
            pack ( [λ_:⟦c⟧.of_unit] ,
                 UnitTerm ⌜1⌝ unit))
 |  typecheck [c] [_] (context, (App ⟨c, 'e1, 'e2⟩,(e1, e2))) =
        let
            pack ('tp12, d1, tp12) = typecheck [c] ['e1] (context, e1)
            pack ('tp2, d2, tp2) = typecheck [c] ['e2] (context, e2)
        in
            case tp12 of
                 Arrow ⟨'tp11,'tp12⟩ (tp11,tp12) =>
                       let
                           pack (d3, unit) = checkEqTp ['tp11] ['tp2] (tp11, tp2)
                       in
                           pack ('tp12,
                                 pack (⌜                                    ⌝ ,
                                      λγ:⟦c⟧.of_app ⟦'tp11⟧ ⟦'tp2⟧ ⟦'tp12⟧
                                                          (⟦'e1⟧γ) (⟦'e2⟧γ)
                                                          ⟦d3⟧ (⟦d2⟧γ) (⟦d1⟧γ)
                                       tp12))
                       end
                | UnitType _ _ => error
        end
 |  typecheck [c] [_] (context, Lam ⟨_,'tp1,'e⟩(tp1, e)) =
        let
            pack ('tp2,d1,tp2) = typecheck [⌜⟦c⟧×Σe1:exp.of e1 ⟦'tp1⟧⌝]
                                                  [⌜λγ:•.⟦'e⟧⟨π₁γ, π₁₂γ⟩⌝]
                                       ((Cons ⟨c,'tp1⟩ (context,tp1)), e)
        in
            pack (⌜arrow ⟦'tp1⟧ ⟦'tp2⟧⌝ ,
                  pack (⌜                                                   ⌝ ,
                       λc1:⟦c⟧.of_lam ⟦'tp1⟧ ⟦'tp2⟧
                                  (λe1:exp.⟦'e⟧(e1))
                                  (λe1:exp. λd2:(of e1 ⟦'tp1⟧).⟦d1⟧⟨c1,e1,d2⟩)
                       Arrow ⟨'tp1,'tp2⟩ (tp1, tp2) ))
        nd
```