Twelf as a unified framework for language formalization and implementation

Rob Simmons
Advisor: David Walker

Princeton University

April 29, 2005

Acknowledgements

I would like to first thank my two independent work advisers: David Walker, whose enthusiasm convinced me to do a senior thesis in the first place and (even more importantly) kept me going throughout, and also Andrew Appel, whose encouragement and faith in me during each of the moments of doubt I experienced while learning Twelf was the only reason I stuck with this past February 2004.

Instrumental to my ability to do this project were the friends and mentors who I had the privilege of working with last summer: Dan Wang, Aquinas Hobor, and Chris Richards. I also could not have done this project without the assistance of Frank Pfenning, Karl Crary, and Susmit Sarkar; finally, I would be remiss not to thank Dan Wang and Dan Peng for their invaluable comments on sections of this thesis.

Special thanks go to the entire community of the Princeton Tower Club, especially the residents and staff who never kicked me out, and to all of those who kept me company in the cluster over the course of this year; without them I could *never* have pulled this off.

Finally, I would like to thank my family and all the members of my "extended village" from Glenn Memorial to ECP. They are the only reason I could dream to do something as wonderfully crazy as going to Princeton to write a senior thesis in the first place.

Sometimes,
life needs to not make sense.

Contents

Chapter 1

Introduction

The adoption of computer checked methods for verifying properties of programming languages is overdue in the programming languages community. We are not the first to express a this sentiment; theorem proving researchers have been declaring since 1998 that "the technology for producing machine-checked programming language designs has arrived" [7]. However, the PL community has been slow to adopt computer-verified means of understanding properties of computer languages. In response to this slow adoption, a group of theorists, primarily at the University of Pennsylvania, announced PoplMark [3], issuing a challenge that involved a machine-checked formalization of a theoretically interesting computer language, System $F_{<:}$.

Nine days later, Karl Crary, Robert Harper, and Michael Ashley-Rollman at Carnegie Mellon announced that they had provisionally completed the challenge, and while parts of their solution remain controversial, their solution, which used similar methods to the ones which form the core of this project, has the potential to increase the adoption of these methods in the wider programming languages community.

This report describes a project aimed at developing the practice of formally establishing the type safety of programming languages using Twelf's metatheory capabilities. From the beginning, the goal was to use Twelf to develop a *realistic* language formalization; a formalization where the abstract syntax might reasonably be generated by a parser, and where the static and dynamic semantics are specified in algorithmic way, so that they could actually be run on the abstract syntax representation of programs. Chapter 2 will present a full formalization of a minimalist imperative language, introducing the methods of this project and the challenges involved; like the rest of this report, it assumes a basic understanding of the foundations of programming language theory. Chapter 3 will consider the extension of this formalization to a more interesting language named Fun, which is defined in Appendix A; furthermore, it will describe the integration of that formalization into a compiler frontend. Chapter 4 concludes with a brief description of other work done as part of the project, as well as with commentary on the future of language formalization, especially in Twelf.

1.1 Formalizing programming languages

At its core, this project deals with writing mathematical proofs about the behavior of programs. When we call this a "formalization" project, we are following Norrish in our understanding that "the term 'formal' implies 'admits logical reasoning'" [8, Section 1.2]. Therefore, the first part of a formalization process is a mathematical specification of a language. This is desirable on its own, as it removes much of the ambiguity inherent in any English description of a computer language, and is therefore a valuable resource for a compiler author.

Because any realistically usable modern computer language will have a relatively large amount of of detail when compared to most theoretical programming languages or logical frameworks, and because much of this detail is often not very theoretically interesting (which is why it is typically excluded from theoretical programming languages), it is difficult to effectively maintain, test, and reason a formalization unless that formalization is in a computer-accessible format.

This computerized formal encoding of the syntax and semantics of a programming language is then useful in other ways as well; the formal reasoning system in which the language is encoded can be used to reason about the language. There are two different approaches to reasoning about the behavior of a computer program.

The first approach, exemplified by Norrish's work on C [8], involves creating proofs about the behavior of individual programs. These approaches often use Hoare logic, which deals with the way that facts about a language change at each step of its execution. These proofs are hard to do automatically, however, and while they can potentially allow many properties of individual programs to be established, doing so typically requires an enormous amount of effort.

The second approach is more common - rather than reasoning about the behavior of individual programs, properties of *all* programs written in a certain language are established. The classic example involves the safety guarantee made by many languages like Java and ML, a promise which is presented in English as statements like "programs that pass the type checker will not go wrong, for instance by accessing private data or crashing the computer."

In the first approach, the goal is essentially to reason about programs written in our formalized language, about the relationship between the language's semantics and *objects*, i.e. the mathematical representations of individual programs. In the second example, the goal is to reason about any program that can be written in a certain language, about the relationship *between* judgements that can be made about programs, judgements such as "the expression e has type `int`" or "the program state e_1 steps to the program state e_2." Because the derivation of a certain judgement can be seen as the proof of a theorem, verified statements concerning the relationships between different judgements are called *metatheorems*. Because these proofs are generally very tedious and complex for realistic full-scale programming languages, proof assisting systems are needed to assist in the proving of these kinds of metatheorems.

1.2 The Twelf system

Twelf is one such system that allows for efficient and natural encoding of the syntax of programming languages, and has powerful capabilities for proving metatheorems about these encodings. Twelf's approach is somewhat unique among modern theorem proving systems. Rather than a single reasoning system, it has two logic systems that coexist with each other. The inner system is an *object logic* using the higher-order abstract syntax of LF, which is just the pure dependently typed lambda calculus. The outer system is a much more powerful *metalogic* which the user interacts with through a logic programming metaphor.

1.2.1 Reasoning in the parametric function space of LF

The dependently typed object logic by itself is very powerful, and can be used AUTOMATH-style to write proofs by hand that can be mechanically verified by a relatively small trusted prover. The object logic is not extraordinarily interesting computationally - the only functions that exist are ones that implement capture-avoiding substitution. This is enough computational power, however, to describe the syntax of most programming languages in a natural way.

More computationally interesting programs can be written using a logic programming metaphor built into Twelf. The author has previously worked on using this logic programming capability to build a tactical prover that automatically generates proofs about facts related to simple arithmetic [14]; however, the current Twelf system's logic programming capabilities are less powerful than the logic programming system used in Appel and Felty's general discussion of these tactical provers [1]. Twelf only allows definition of what can be thought of as constructive, recursively defined functions.

1.2.2 Reasoning in the recursive function space of M_2^+

There is an extremely good reason for Twelf limiting the computational abilities of its logic programming capabilities. Proving metatheorems about the relationship between various judgements requires that the function space of those judgements be well defined; more computationally interesting systems, for instance those that allow the cut operator for fine-tuned control over backtracking, have a function space that is too complex and cannot be reasoned about in a metatheoretic way [10, Section 5.8].

The function space of Twelf is referred to as M_2^+, and while it is limited in many ways, it is quite adequate in most cases for both describing the semantics of programming languages and for writing metatheorems about those semantics. Proving metatheorems in Twelf in the style used by this project is roughly equivalent to writing a detailed induction proof by cases - the assistance Twelf gives is in examining the purported inductive proof and giving one of two responses, either "yes, this represents a metatheorem," or "no, this purported metatheorem is incomplete for the following (potentially spurious) reasons..." The information as to why proofs are incomplete is one of the most useful aspects of the theorem-proving environment.

1.2.3 The (once and future) Twelf theorem prover

Twelf also includes an automated theorem prover, which takes the declaration of a metatheorem and returns whether or not it is able to establish that metatheorem by induction. In its original form, it returned the inductive proof by cases that the previous section describes being written by hand.

This ability was disabled with the release of Twelf 1.4; due to this, and to the fact that the prover in general is tagged as "even more experimental" than the rest of Twelf [10, Chapter 10], and to the theorem prover's current inability to produce the useful error messages that are generated when writing inductive proofs by hand, the theorem prover was completely ignored by this project.

1.3 The language of type checkers

Almost all other theorem provers use first-order abstract syntax, and thus have significant issues with dealing with binders [3], leading to an enormous and mostly uninteresting proof burden dealing with problems related to capture-avoiding substitution. The essential problem may be simply that they are not the right tool for the job; indeed, many were designed for hardware verification and not for programming language formalization [16].

The type systems that emerge in these systems are often unnatural, and the formalizations of type safety are often not the usual "safety = progress + preservation" style proofs, instead relying on indexed types or other unfamiliar additions. These approaches are useful in many ways, and there is ongoing research into creating more natural formalizations without resorting to Twelf-style metatheorems [19]. However, we believe that while there probably is a place for these more "foundational" approaches to type safety, there is a unique opportunity made available through the Twelf metatheorem prover to write language specifications simply, realistically, and productively, satisfying the PoplMark goal of bringing "mechanized metatheory to the masses" [3].

Regular expressions are the language of parsers, context-free grammar is the language of lexers, and inductive definitions in higher-order abstract syntax are probably the best candidate for the language of type checkers [18]. In a conversation with Aquinas Hobor around the time of this project's inception, he expressed joking concern that the result of this project would be yet another specialized system to add to several that complier writers already have to learn. He was probably right to be concerned; we believe that the strategies described by this paper may represent the precursors to specialized languages for describing and generating type checkers directly from formal type system specifications, specifications which can also be reasoned about by the language's designers to establish type safety and other properties of the language.

1.4 The view from outside

The fast completion of PoplMark by two professors and an undergraduate at Carnegie Mellon follows the unsurprising tendency for the primary users, or at least the early adopters, of theorem proving systems to be in close connection with the developers of those systems. We believe that this project represents, if not the largest, then surely one of the largest language formalizations created outside of Carnegie Mellon and the Twelf developer community; it is also one of what to our knowledge is a very small number of attempts to gain a practical understanding of Twelf's metatheory capabilities without a significant understanding of Twelf's mathematical foundations.

We believe that this outsiders' perspective makes this project a significant contribution to the knowledge of how Twelf metatheory is learned, and it has led to a number of responses. First, we recognized a need for an educational resource for Twelf beyond the ones currently available. The primary resources available to beginning users of Twelf are the Twelf User's Guide [10] and two mailing lists for users and developers of Twelf. The User's Guide is substantially more devoted to system description than to troubleshooting and problem solving, and while mailing lists are often helpful, the archives are not an organized information source and do not always follow the description of a problem through to its resolution. With the help of Twelf co-designer Frank Pfenning we established The Twelf Wiki, aimed at providing a resource for users outside of the developer community that is more flexible than the ones currently available. While most of its material is currently descriptions of problems encountered in the course of this project and solutions to those problems, the Wiki is coming to be utilized by other members of the non-CMU Twelf community, receiving contributions from Stephen Tse at the University of Pennsylvania and Todd Wilson at CSU Fresno.

The second response is this report, which is substantially dedicated to a description of the choices made and challenges encountered in the course of this year-long project. In particular, the second chapter is designed as a crash course in Twelf metatheory as well as an account of the methods utilized this project; it is furthermore an experiment in using Twelf code as a readable description of inductive proofs. Finally, we presented some of the methods used in this project to the programming languages community at Princeton through the TACL seminar series on April 27, 2005.

Chapter 2

Formalizing safe languages in Twelf

This chapter presents a formalization of a simplified version of the Fun language which is discussed in the next chapter and defined in Appendix A. This presentation will discuss not only the mathematical description of the language and safety proof, but also the Twelf encoding of that description. The first section discusses the logical encoding of the example language's syntax and semantics, and the second section introduces metalogic proofs and walks through the metatheorem asserting the type safety of the example language. The chapter concludes by taking a look at how the type checker can report failure information.

This chapter is not intended to be simply a commentary on the code presented, but also an introduction to the concepts involved in the development of that code. It will present many Twelf concepts informally by explaining them in terms of equivalent concepts in the theory of programming languages, and we hope that this approach will allow this chapter to serve as a practical introduction and tutorial for someone with an understanding of programming language theory but without a background Twelf metatheory.

2.1 Encoding syntax and semantics

This section, with one exception (lists of expressions), will present all of the Twelf code needed to formalize the minimalist Fun. Rather than explaining the code itself, it will be presented side by side with the equivalent mathematical representation, which itself serves as an explanation of the Twelf code.

2.1.1 Representing primitive operations

Most languages do useful work by manipulating primitive data types, for instance 32-bit numbers, strings, or objects. Twelf supports "importing" strings, infinite-precision integers, and 32-bit unsigned integers into the pure LF framework, but in the current implementation doing so is completely incompatible with Twelf's ability to prove metatheorems - misusing the 32-bit unsigned integer package, for instance, can in some cases result in Twelf appearing to assert the truth of untrue metatheorems [15, Constraint Domains].

Because Twelf's "built in" integers and strings cannot currently be safely or usefully used in a language formalization process that involves a safety proof, a different approach is needed; there are essentially two options. The first is to have a single "dummy" value which stands in for all of the possible values of a given primitive type. This is perfectly fine from the perspective of typechecking, but if the language has control flow capabilities, such as `while` or `if`, which behave differently based on the value of a primitive data type, then this approach will result in a specification of the dynamic semantics which does not match the language's actual behavior; however, the dynamic semantics

Natural numbers: $n ::= \mathsf{z} \mid \mathsf{s}(n)$

$$\frac{}{\mathsf{z} + n \Downarrow n}\ \texttt{sum-z}$$

$$\frac{n_1 + n_2 \Downarrow n_3}{\mathsf{s}(n_1) + n_2 = \mathsf{s}(n_3)}\ \texttt{sum-s}$$

```
nat : type.
z : nat.
s : nat -> nat.

sum : nat -> nat -> nat -> type.
%mode sum +A +B -C.

sum-z : sum z N N.
sum-s : sum (s N1) N2 (s N3)
           <- sum N1 N2 N3.
%worlds () (sum A B C).
%total A (sum A B C).
```

Encoding 1: The natural numbers

could still be simulated in a nondeterministic way - for instance, if there was only a single integer, then a `while` loop that branched according to whether that integer was 0 could be specified as branching each of the two possible ways.

The second approach is to create (or find) a formalization of whatever primitive data types the language manipulates. This solution has much more overhead, but it is crucial if using the specification as a “runnable” simulation of the language is a goal. This project uses a binary number specification based on work done by the TALT project, but the formalization in this chapter will use the natural number specification (Encoding 1) discussed in more detail by Crary [4].

The construction of `nat` as zero and successors of other natural numbers should be clear, but a couple of important points are highlighted by `sum`. In the context of this section, we will view `sum` as a *logic program* which can be executed. In this view, the `%mode` declaration checks that `sum` can be thought of as implementing a partial function with two inputs (`+A` and `+B`) and one output (`-C`). `%worlds` and `%total` will be discussed later in relation to metatheorems, but for now it suffices to say that they cause Twelf to check that the partial funciton `sum` is in fact a total function, i.e. a logic program that accepcts all inputs and always sucessfully terminates after a finite number of steps. The first `A` in the `%total` declaration gives Twelf the hint it needs to determine why `sum` always terminates: the first argument, `A`, gets smaller with each recursive call to `sum`, so if the input has finite size, then `sum` cannot keep making recursive calls forever.

Finally, Encoding 1 demonstrates the crucial point that identifiers starting with uppercase letters represent metavariables. Twelf reconstructs the type of all metavariables (for instance, `N` in the definition of `sum-z` has type `nat`) and allows the metavariable to represent any derivable object with that type.

2.1.2 Syntax: types and expressions

The language used here is intended to be the smallest that is useful for discussing all the interesting aspects of the Fun formalization. As such, it has `let` statements, the ability to create and dereference references, and natural numbers with an addition operator. There is no assignment operation only for brevity's sake - assignment does not pose any especially interesting theoretical or practical problems, but assignment *is* included in the full formalization. To the definition of natural numbers, Encoding 2 adds the expressions and types of the minimalist Fun language.

Except for `let`, all the constructs in the syntax encoding should be fairly self-explanatory. Most of the expression constructors take one or more objects with a certain type (for instance, `ref` takes an expression) and construct an expression. Twelf also has the ability to represent infix operators with left, right, or no associativity, which `+` demonstrates.

$$\begin{array}{rrcl}
\text{Locations:} & l & ::= & n \\
\\
\text{Expressions:} & e & ::= & \mathsf{n}(n) \mid e + e \\
 & & & \mid \mathsf{ref}(e) \mid \mathsf{!}e \mid \mathsf{loc}(l) \\
 & & & \mid \mathsf{let}\ x{=}e\ \mathsf{in}\ e \\
\\
\text{Types:} & \tau & ::= & \mathsf{int} \mid \mathsf{ref}(\tau)
\end{array}$$

```
%abbrev location = nat.

exp : type.
n : nat -> exp.
+ : exp -> exp -> exp. %infix left 10 +.
ref : exp -> exp.
! : exp -> exp.
loc : location -> exp.
let : exp -> (exp -> exp) -> exp.

tp : type.
int-tp : tp.
ref-tp : tp -> tp.
```

Encoding 2: Basic syntax of the toy language

What makes `let` more interesting is that the second object it expects has type `(exp -> exp)`. This indicates that there will be a λ-binding, which is introduced using the bracket symbols. Take, for example, these three definitions:

```
foo1 : exp -> exp = [x] x + x + (n (s z)).
foo2 : exp = let (n z) ([x] x + x + (n (s z))).
foo3 : exp = foo1 (n z).
```

The definition of `foo1` can be read as "`foo1` has type `exp -> exp` and is the object `[x] x + x + (n (s z))`," and it demonstrates using brackets to introduce the λ-bound variable `x`. `foo2` is the Twelf encoding of the expression $\mathsf{let}\ x{=}\mathsf{n}(\mathsf{z})\ \mathsf{in}\ x + \mathsf{n}(\mathsf{s}(\mathsf{z}))$, and `foo3` represents how an expression `(n z)` is applied to `foo1`; Twelf sees `foo3` as equivalent to the expression `(n z) + (n z) + (n (s z))`.

The ability to represent binders for the underlying language as a binders in the higher-order abstract syntax, as this example illustrates, was cited by the designers of PoplMark as one of the greatest strengths of using a higher-order abstract syntax implementation like Twelf for formalization, though the approach has significant limits, such as representing lists of mutually recursive binders. Other limits include the apparent inability to specify a closure conversion algorithm, or to establish certain properties such as strong normalization, which are not dealt with by this project [3].

2.1.3 Lists of types and expressions

Notice from the previous section that a location in this language is simply an abbreviation for a natural number. We define the store and store typing as lists of expressions and types. Dereferencing is implemented as projection, so that dereferencing the pointer $\mathsf{loc}(n)$ will be done by accessing the n^{th} element in the list which represents the store. Allocation will append new values to the end of the list. Unfortunately, Twelf does not offer any kind of list polymorphism, and so each list is a new data type - in the larger specification where several lists needed similar capabilities, a single "master" file was written that introduced the type `any` and lists of `any`s. When a new kind of list was needed, say lists of `tp`s, an automatic find/replace function renamed all instances of `any` in the master file with `tp`, and the result was saved as a new file. Synchronization was obviously an issue, but as the requirements on lists stayed relatively stable, it was not a major problem. Encoding 3 describes lists of types, which can be converted into the encoding for lists of expressions by replacing all instances of `tp` with `exp` and, optionally, all instances of `T` with `E`.

$$\begin{array}{rrcl} \text{Lists of types:} & \tau_{\text{list}} & ::= & \mathsf{nil} \mid \tau :: \tau_{\text{list}} \\ \text{Lists of expressions:} & e_{\text{list}} & ::= & \mathsf{nil} \mid e :: e_{\text{list}} \end{array}$$

```
tplist : type.
$tp : tp -> tplist -> tplist.
%infix right 5 $tp.
nil-tp : tplist.
```

$$\frac{}{(\tau :: \tau_{\text{list}}).\mathsf{z} \Downarrow \tau}\ \texttt{proj-tp-z}$$

$$\frac{(\tau_{\text{list}}).n \Downarrow \tau}{(\tau' :: \tau_{\text{list}}).\mathsf{s}(n) \Downarrow \tau}\ \texttt{proj-tp-s}$$

```
proj-tp : tplist -> nat -> tp -> type.
%mode proj-tp +TL +N -T.
proj-tp-z : proj-tp (T $tp TL) z T.
proj-tp-s : proj-tp (T $tp TL) (s N) T'
             <- proj-tp TL N T'.
```

$$\frac{}{\mathsf{nil}\ @\ \tau \Downarrow (\tau :: \mathsf{nil}\ ,\ \mathsf{z})}\ \texttt{append-tp-z}$$

$$\frac{\tau_{\text{list}}\ @\ \tau \Downarrow (\tau'_{\text{list}}\ ,\ n)}{\tau' :: \tau_{\text{list}}\ @\ \tau \Downarrow (\tau' :: \tau'_{\text{list}}\ ,\ \mathsf{s}(n))}\ \texttt{append-tp-s}$$

```
append-tp : tplist -> tp
             -> tplist -> nat -> type.
%mode append-tp +TL +T -TL' -N.
append-tp-z : append-tp nil-tp T
               (T $tp nil-tp) z.
append-tp-s : append-tp (T' $tp TL) T
               (T' $tp TL') (s N)
               <- append-tp TL T TL' N.
```

$$\frac{}{\mathsf{nil} \subseteq \tau_{\text{list}}}\ \texttt{subset-tp-z}$$

$$\frac{\tau_{\text{list}} \subseteq \tau'_{\text{list}}}{\tau :: \tau_{\text{list}} \subseteq \tau :: \tau'_{\text{list}}}\ \texttt{subset-tp-s}$$

```
subset-tp : tplist -> tplist -> type.
subset-tp-z : subset-tp nil-tp T.
subset-tp-s : subset-tp (T $tp TL)
                         (T $tp TL')
               <- subset-tp TL TL'.
```

Encoding 3: Lists of types

$$\begin{array}{rrcl} \text{Store:} & \mu & ::= & e_{\text{list}} \\ \text{Store typing:} & \Sigma & ::= & \tau_{\text{list}} \end{array}$$

```
%abbrev store = explist.
%abbrev storetp = tplist.
```

Encoding 4: Store abbreviations

Encoding 3 also demonstrates the ways in which the mathematical notation will appear to use polymorphism in some cases (for instance, the apparently polymorphic or overloaded cons (::) operator) when such a usage makes the notation simpler, even though this does not perfectly match the representation in Twelf.

2.1.4 Shortcuts for the store

The store and store typing may just be a list of expressions and types, but in order for the rest of the formalization to be clearer we will use the symbol μ for the store and Σ for the store typing, following Pierce [11, Chapter 13]. Encoding 4 shows this and the related abbreviations in Twelf.

2.1.5 Values

The last piece of the puzzle before we define the semantics of the language is a judgement defining which expressions are values. Values are here defined as a subset of expressions that satisfy a certain

$$\frac{}{\mathsf{isval}(\mathsf{n}(n))}\ \texttt{v-int}$$

$$\frac{}{\mathsf{isval}(\mathsf{loc}(l))}\ \texttt{v-loc}$$

$$\frac{}{\mathsf{isval}(\mathsf{nil})}\ \texttt{vl-z} \qquad \frac{\mathsf{isval}(e) \quad \mathsf{isval}(e_{\text{list}})}{\mathsf{isval}(e :: e_{\text{list}})}\ \texttt{vl-s}$$

```
isval : exp -> type.
%mode isval +E.
v-int : isval (n N).
v-loc : isval (loc L).

isval-list : explist -> type.
%mode isval-list +E.
vl-z : isval-list nil-exp.
vl-s : isval-list (E $exp EL)
       <- isval E
       <- isval-list EL.
```

Encoding 5: Values

predicate (Encoding 5).

2.1.6 Dynamic semantics

Now all the pieces necessary to define the language's dynamic syntax are in place (Encoding 6). The program state is a pair of a store and an expression.

Notice that the operation of `e-let` is to apply the expression `E` to the higher-order function `EF`, which could have equivalently been represented in its *eta-expanded* form `([x] EF x)`. As it was demonstrated in Section 2.1.2, this will replace all instances of `EF`'s bound variable with `E`.

2.1.7 Static semantics

It was mentioned in Section 2.1.2 that one of the great benefits of formalizing a language in a system like Twelf is that it allowed for binders to be introduced into the language using binders from the metalogic. The typing relation demonstrates another huge convenience: Twelf in some sense provides the context Γ "for free."

When the typing derivation reaches `t-let`, it adds to the LF context a fresh expression `x`, about which the only derivable statement is `var-typed x T` for some already-determined `T`. If we view `typed` as a logic program which is determining the type of the expression by recursion, then what is happening in the second part of `t-let` is that that recursion is "traversing the λ-binding," since the metavariable EF represents some object with type `(exp -> exp)` - a fact which will later have important consequences for parts of the safety proof. The only way that Twelf will successfully determine the type of an introduced expression `x` is by attempting `t-var`, which will check the LF context, find the simultaneously introduced `var-typed x T`, and will thus derive `typed ST E T` for whatever storetype `ST` is currently being used.

This approach requires the use of a substitution lemma later on - it is worth noting that where there is no store or store typing, the assumption `typed x T` can itself be placed into the LF context for the introduced expression `x`, which makes the substitution lemma unnecessary. Introducing a store typing to the `typed` judgement, however, prevents this method from being used. Trying to introduce into the context a judgement `typed ST x T` would make it impossible to prove the store type weakening lemma that is needed later on [15, Worlds and Weakening].

2.1.8 Running a logic program

It was previously mentioned that `sum`, `typed`, `proj-tp`, and `eval` all represented logic programs or partial functions, without explaining what this actually means. Without going into too much detail as to how Twelf works internally, consider the encoding of "$\mathsf{let}\ x{=}\mathsf{ref}(\mathsf{n}(\mathsf{s}(\mathsf{z})))\ \mathsf{in}\ \mathsf{n}(\mathsf{s}(\mathsf{z})) + !x$":

$$\frac{\mu \mid e_1 \longrightarrow \mu' \mid {e_1}'}{\mu \mid e_1 + e_2 \longrightarrow \mu' \mid {e_1}' + e_2}\ \texttt{s1-add}$$

$$\frac{\mathsf{isval}(e_1) \quad \mu \mid e_2 \longrightarrow \mu' \mid {e_2}'}{\mu \mid e_1 + e_2 \longrightarrow \mu' \mid e_1 + {e_2}'}\ \texttt{s2-add}$$

$$\frac{n_1 + n_2 \Downarrow n_3}{\mu \mid \mathsf{n}(n_1) + \mathsf{n}(n_2) \longrightarrow \mu \mid \mathsf{n}(n_3)}\ \texttt{e-add}$$

$$\frac{\mu \mid e \longrightarrow \mu' \mid e'}{\mu \mid \mathsf{ref}(e) \longrightarrow \mu' \mid \mathsf{ref}(e')}\ \texttt{s-ref}$$

$$\frac{\mathsf{isval}(e) \qquad \mu \mathbin{@} e \Downarrow (\mu' , l)}{\mu \mid \mathsf{ref}(e) \longrightarrow \mu' \mid \mathsf{loc}(l)}\ \texttt{e-ref}$$

$$\frac{\mu \mid e \longrightarrow \mu' \mid e'}{\mu \mid !e \longrightarrow \mu' \mid !e'}\ \texttt{s-bang}$$

$$\frac{(\mu).l \Downarrow e}{\mu \mid !\mathsf{loc}(l) \longrightarrow \mu \mid e}\ \texttt{e-bang}$$

$$\frac{\mu \mid e_1 \longrightarrow \mu' \mid {e_1}'}{\mu \mid \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 \longrightarrow \mu' \mid \mathsf{let}\ x{=}{e_1}'\ \mathsf{in}\ e_2}\ \texttt{s-let}$$

$$\frac{\mathsf{isval}(e_1)}{\mu \mid \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 \longrightarrow \mu \mid [x \mapsto e_1]e_2}\ \texttt{e-let}$$

```
eval : store -> exp
    -> store -> exp -> type.
%mode eval +S +E -S' -E'.

s1-add : eval S (E1 + E2) S' (E1' + E2)
          <- eval S E1 S' E1'.
s2-add : eval S (E1 + E2) S' (E1 + E2')
          <- isval E1
          <- eval S E2 S' E2'.
e-add : eval S (n N1 + n N2) S (n N3)
         <- sum N1 N2 N3.

s-ref : eval S (ref E) S' (ref E')
         <- eval S E S' E'.
e-ref : eval S (ref E) S' (loc L)
         <- isval E
         <- append-exp S E S' L.

s-bang : eval S (! E) S' (! E')
         <- eval S E S' E'.
e-bang : eval S (! (loc L)) S E
         <- proj-exp S L E.

s-let : eval S (let E EF) S' (let E' EF)
         <- eval S E S' E'.
e-let : eval S (let E EF) S (EF E)
         <- isval E.
```

Encoding 6: Dynamic semantics

```
prog = let (ref (n (s z))) [x] (n (s z)) + (! x).
```

We can determine the result of using the logic program `typed` on the empty store typing and `prog` by loading this statement:

```
%solve type-test : typed nil-tp prog T.
```

Twelf responds by replacing the metavariable `T` with the result of running the logic program, the type of `prog`, i.e. `int-tp`. This is Twelf's full response:

```
type-test : typed nil-tp prog int-tp
   = t-let
        ([x:exp] [x1:var-typed x (ref-tp int-tp)]
            t-add (t-bang (t-var x1)) t-int) (t-ref t-int).
%% OK %%
```

However, something else is happening when Twelf succeeds in executing the logic program. It has defined `type-test`, which is itself an object with *type* `(typed nil-tp prog int-tp)` and

$$\frac{}{\Gamma \mid \Sigma \vdash \mathsf{nil} : \mathsf{nil}}\ \texttt{tl-z}$$

$$\frac{\Gamma \mid \Sigma \vdash e : \tau \quad \Gamma \mid \Sigma \vdash e_{\text{list}} : \tau_{\text{list}}}{\Gamma \mid \Sigma \vdash (e :: e_{\text{list}}) : (\tau :: \tau_{\text{list}})}\ \texttt{tl-s}$$

$$\frac{}{\Gamma \mid \Sigma \vdash \mathsf{n}(n) : \mathsf{int}}\ \texttt{t-int}$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \mathsf{int} \quad \Gamma \mid \Sigma \vdash e_2 : \mathsf{int}}{\Gamma \mid \Sigma \vdash e_1 + e_2 : \mathsf{int}}\ \texttt{t-sum}$$

$$\frac{\Gamma \mid \Sigma \vdash e : \tau}{\Gamma \mid \Sigma \vdash \mathsf{ref}(e) : \tau\ \mathsf{ref}}\ \texttt{t-ref}$$

$$\frac{\Gamma \mid \Sigma \vdash e : \tau\ \mathsf{ref}}{\Gamma \mid \Sigma \vdash\ !e : \tau}\ \texttt{t-bang}$$

$$\frac{(\Sigma).l \Downarrow \tau}{\Gamma \mid \Sigma \vdash \mathsf{loc}(l) : \tau\ \mathsf{ref}}\ \texttt{t-loc}$$

$$\frac{\Gamma \mid \Sigma \vdash e_1 : \tau' \quad \Gamma, x : \tau' \mid \Sigma \vdash e_2 : \tau}{\Gamma \mid \Sigma \vdash \mathsf{let}\ x{=}e_1\ \mathsf{in}\ e_2 : \tau}\ \texttt{t-let}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \mid \Sigma \vdash x : \tau}\ \texttt{t-var}$$

```
var-typed : exp -> tp -> type.
%mode var-typed +E -T.
typed : storetp -> exp -> tp -> type.
%mode typed +ST +E -T.

typed-list : storetp -> explist
                     -> tplist -> type.
%mode typed-list +ST +E -T.
tl-z : typed-list ST nil-exp nil-tp.
tl-s : typed-list ST (E $exp EL)
                     (T $tp TL)
        <- typed ST E T
        <- typed-list ST EL TL.

t-int : typed ST (n N) int-tp.
t-add : typed ST (E1 + E2) int-tp
         <- typed ST E1 int-tp
         <- typed ST E2 int-tp.
t-ref : typed ST (ref E) (ref-tp T)
         <- typed ST E T.
t-bang : typed ST (! E) T
          <- typed ST E (ref-tp T).
t-loc : typed ST (loc L) (ref-tp T)
         <- proj-tp ST L T.
t-let : typed ST (let E EF) T
         <- typed ST E T'
         <- ({x} var-typed x T'
               -> typed ST (EF x) T).
t-var : typed ST E T
         <- var-typed E T.
```

Encoding 7: Static semantics

which is derivable as `(t-let ([x:exp][x1:var-typed x (ref-tp int-tp)]...))`. This touches on many of the complicated features of LF, such as its dependent type system and *judgments-as-types* representation technique, which will not be considered here as they have been thoroughly discussed elsewhere [11, Section 30.5][9]. The relevant point is that it makes sense to think of `sum` and `typed` as logic programs, but in some contexts it is useful to think of them in terms of derivation trees, which are *proof objects* that may themselves be manipulated by logic programs in the same way that $(\mathsf{s}(\mathsf{s}(\mathsf{z})))$ is an object which may be manipulated by `sum`.

2.1.9 Muti-step evaluation

The definition of the Kleene closure of the step relation (Encoding 8) shows the importance of the difference between the logic programming metaphor and proof object metaphor. All the previously described logic programs, such as `typed` and `eval`, are (probably) deterministic[1], meaning that there is only one rule that actually needs to be considered for any set of inputs. This makes `typed` and

[1]We say "probably" simply because this is never formalized

$$\frac{\mu \mid e \longrightarrow \mu' \mid e' \qquad \mu' \mid e' \longrightarrow^* \mu'' \mid e''}{\mu \mid e \longrightarrow^* \mu'' \mid e''}\ \texttt{run-step}$$

$$\frac{}{\mu \mid e \longrightarrow^* \mu \mid e}\ \texttt{run-end}$$

```
run : store -> exp
   -> store -> exp -> type.
%mode run +S +E -S' -E'.

run-step : run S E S'' E''
            <- eval S E S' E'
            <- run S' E' S'' E''.
run-end : run S E S E.
```

Encoding 8: $\longrightarrow^*$

`eval` (probably) behave much as if they were performing case evaluation using pattern matching in a language like ML. However, when executing logic programs, Twelf will always attempt to use the first-defined option first, so that if `run` is executed as a logic program it will always attempt to derive an evaluation step using `run-step` until the derivation of an additional evaluation step cannot be found, and then `run-end` will be attempted and will always succeed. In other words, `run` will run the program until it cannot run anymore.

However, as a specification of a derivation tree, `run`/$\longrightarrow^*$ represents *any* amount of evaluation, as shown by the very simple expression `prog2` below, with two different derivations of `run` that show evaluation running for 0 and 1 steps, respectively.

```
prog2 = (n z) + (n (s z)).

run1 : run nil-exp prog2 nil-exp prog2 = run-end.
run2 : run nil-exp prog2 nil-exp (n (s z)) = run-step run-end (e-add sum-z).
```

Running the logic program `run` would first determine the second derivation, but both are valid proof objects establishing two different judgements. In fact, with backtracking `run` is able to generate both derivations, but this is not a capability that this project utilizes.

2.2 Proving language safety

The end of the last section discussed logic programming and the ability of a logic program to return a proof object representing the derivation of the result. In this section we begin to write relations that manipulate those objects, and consider some of the consequences in the context of developing the safety proof of the minimalist Fun language that was just presented.

The theory behind this section is noncontroversial and is covered thoroughly by Pierce [11, Chapter 13], and this presentation follows Pierce's as closely as possible. Because this traditional presentation of an extremely similar safety proof is available, and also because inductive proofs are not typically represented in a rigorous symbolic format in the way that inductive definitions are, this section will abandon the side by side mathematical specification used in the previous section and begin to let the Twelf code speak for itself.

This approach was considered by Syme, who considered readability by outsiders to be an extremely important element of a formalization system [16], and also by Crary, who presented many of the lemmas leading to the safety proof for TALT in their Twelf form [4], but this section will attempt to use code as an explanatory tool even more extensively. This section is long, and some parts of the proof are tedious, because inductive proofs are, in fact, generally long and tedious. Trivial and uninteresting lemmas will therefore generally be declared, but their proofs will be omitted from this paper; furthermore, the commentary on most lemmas will also serve to discuss a new issue related to proving metalemmas in Twelf.

2.2.1 Introduction to metatheorems & effectiveness of primitive operations

We will start with a noncontroversial example, belonging to a class of lemmas that, following the TALT project, will be called *effectiveness lemmas*. Because `%total A (sum A B C)` passed, we know that for any grounded inputs `sum` will produce a grounded output in a finite number of steps. The first metalemma we will discuss proves a very similar fact which Twelf nevertheless sees as different - that for any two natural numbers `A` and `B`, it is *possible to derive* a proof object `sum A B C` (for some `C`).

We do this by first defining a partial function, `can-sum`, from a pair of natural numbers `A` and `B` to a derivation of `sum A B C`, which looks like this:

```
can-sum : {A: nat}{B: nat} sum A B C -> type.
%mode can-sum +A +B -SUM.
```

This definition of `can-sum` looks significantly different that the definition of `sum` and the other previously defined logic programs, but the only difference is that two inputs are given names here. As a clarifying example, these two definitions of `sum` are seen by Twelf as exactly equivalent:

```
sum : nat -> nat -> nat -> type.
sum : {A: nat}{B: nat}{C: nat} type.
```

Similarly, we could have defined `can-sum` this way:

```
can-sum : {A: nat}{B: nat}{SUM: sum A B C} type.
```

The point is that, in practice, it is more clear not to assign names to inputs when it is unnecessary, but in the case of `can-sum`, identifiers need to be attached to the two natural numbers so that they can be referred to later in the declaration.

Consider for a moment how one would prove, by a standard long-form proof of induction, that $n_1 + n_2 \Downarrow n_3$ is derivable for any two natural numbers n_1 and n_2. The proof, in its most verbose form, might look something like this:

> The proof of the effectiveness of sum is carried out by induction on the structure of the first natural number. If the first natural is `z`, then the entire derivation of the sum is simply `sum-z`, and we are done. If the first natural is $\texttt{s}(n)$, then the second input is some n_2 and the inductive step gives us $n + n_2 \Downarrow n_3$. Applying `sum-s` gives us the derivation we desire, $\texttt{s}(n) + n_2 \Downarrow \texttt{s}(n_3)$. There are no other cases, so we are done.

Now consider again the logic program `can-sum`. Right now it is a partial function with no solutions. However, if we could define the program in such a way that we could prove that `can-sum`, like `sum`, is a *total* function, it would mean that we have specified a recursive logic program that will always succeed in constructing the derivation of `sum` from any two inputs.[2] This is the essence of a metatheorem, the idea that it is possible to prove a theorem of the form “for all A there exists a B” by creating a recursive logic program from objects representing A to objects representing B, and then using the coverage checker to prove that the function is total, so that for any inputs A to the function we can construct in a finite number of steps (and therefore “there exists”) a B.

The `%total` declaration completes the statement of this metatheorem. The totality check requires that there be an ordering which confirms that the proof will always terminate (the `T` in `can-sum T _ _`). If Twelf can confirm termination, it checks coverage, making sure that all possible input cases are covered. Finally, it checks to make sure that no constraints are put on any outputs. Taken together, totality, input coverage, and output coverage complete the verification of the metatheorem.

[2]The totality of sum that we already established in Section 2.1.1 is actually verifies a *stronger* property, that this derivation exists for all inputs *and* will always be found by a prolog search in a finite number of steps given grounded inputs. However, that is why this proof isn't particularly controversial.

```
can-sum : {A: nat}{B: nat} sum A B C -> type.
%mode can-sum +A +B -SUM.
& : can-sum z N sum-z.
& : can-sum (s N1) N2 (sum-s SUM)
     <- can-sum N1 N2 SUM.
%worlds () (can-sum _ _ _).
%total T (can-sum T _ _).
```

Note that because we do not want to prove what Crary calls meta-meta-theorems, the labels for the cases of individual inductive proofs are superfluous and can all replaced with the `&` symbol [4, Section 2.3]. Put another way, we're not interested in the proof objects of these programs - in fact, there is really no need to ever run them, as the important fact is only that they can always successfully *be* run.

In the safety proof for our example language there is another effectiveness lemma, which establishes that for any list of expressions that we might have, a new expression can always be appended to the end of the list. The proof proceeds by induction on the structure of `EL`, the list of expressions, but the proof will be omitted here.

```
can-append-exp : {EL}{E} append-exp EL E EL' N -> type.
%mode can-append-exp +EL +E -APPEND.
```

A more interesting example, and the last lemma we need before proving the progress theorem, is `typed-projection`. The metatheorem proves that if an expression e_{list} has type τ_{list}, then given a derivation of $(\tau_{\text{list}}).n \Downarrow \tau$, a derivation of $(e_{\text{list}}).n \Downarrow e$ can always be constructed for some e that will have type τ.

```
typed-projection : typed-list ST EL TL
                    -> proj-tp TL N T
                    -> proj-exp EL N E
                    -> typed ST E T -> type.
%mode typed-projection +TL +PT -PE -T.
& : typed-projection (tl-s TL T) proj-tp-z proj-exp-z T.
& : typed-projection (tl-s TL T') (proj-tp-s PT) (proj-exp-s PE) T
     <- typed-projection TL PT PE T.
%worlds () (typed-projection _ _ _ _). %total PT (typed-projection _ PT _ _).
```

The first thing that makes this interesting is that is the first proof that has proceeded by induction on what we consider a derivation ($(\tau_{\text{list}}).n \Downarrow \tau$) rather than on what we consider an object (again, from LF's perspective both are just objects). This example also illustrates a quirk of the notation used for logic programming. When *tl-s* was defined, the subgoal concerning the `typed` branch was considered first and the `typed-list` subgoal was listed second - here, `TL` represents the derivation tree for the second subgoal and comes first - the ordering is *reversed*, an occasionally confusing side effect of the backward arrows used in the inductive definitions.

What makes `typed-projection` uniquely interesting, however, is that it will be used for the progress theorem in this form, but for the preservation lemma it will be used as a *different* metalemma which proves that if an expression e_{list} has type τ_{list}, and if we can derive $(\tau_{\text{list}}).n \Downarrow \tau$ *and* $(e_{\text{list}}).n \Downarrow e$, then e will have type τ. After the proof of progress is completed, we simply declare a new mode for the logic program (i.e. metalemma) which we have already shown to be total:

```
%mode typed-projection +TL +PT +PE -T.
```

2.2.2 Progress

The declaration of the proof of progress looks like this:

```
progress : typed ST E T
           -> typed-store ST S
           -> notstuck S E -> type.
%mode progress +T +TS -NS.
```

However, this is not yet a useful declaration, as there is no definition of `typed-store` or `notstuck`. `notstuck` simply expresses that there are two different ways for this language to be not stuck - either the program state can step to a new program state, or the expression is a value.

```
notstuck : store -> exp -> type.
ns-steps : notstuck S E
           <- eval S E S' E'.
ns-isval : notstuck S E
           <- isval E.
```

`typed-store` is even simpler than `notstuck` - it can only be satisfied in one way, with `&typed-store`, which requires that all the expressions in the list be values with a type predicted by the store typing. Its primary purpose is to abstract away, in most cases, the details of the store, i.e. the fact that it is just a list of expressions. Recall, considering the code below, that `storetp` is just an abbreviation for `tplist` and `store` is just an abbreviation for `explist`.

```
typed-store : storetp -> store -> type.
&typed-store : typed-store TL EL
                <- isval-list EL
                <- typed-list TL EL TL.
```

The proof of progress proceeds by induction and case analysis on the last step in the typing derivation, and the different typing derivations will be considered in the order that they were presented (with the exception that `t-ref` will be considered before `t-add`). The first case is simple - progress is immediate in case `t-int` because integers are values. The _ in the input position just means that the second input, i.e. the store typing derivation, is irrelevant to the output.

```
prog-int : progress t-int _ (ns-isval v-int).
```

Skipping over `t-add` momentarily, consider `t-ref`. This is how a written description of this case might look: "From the inductive step, we know that the subexpression is not stuck. If the subexpression can take a step, then progress follows from `s-ref`. If the subexpression is a value, then progress follows from `e-ref` and the expressiveness of the `append-exp` operation guarantees that we can append a new expression to the end of the expression list. There are no other ways for the subexpression to be not stuck, so we are done." One of the logical ways to express this in Twelf is like this:

```
prog-ref1 : progress (t-ref T) TS (ns-steps (s-ref E))
             <- progress T TS (ns-steps E).
prog-ref2 : progress (t-ref T) TS (ns-steps (e-ref APPEND V))
             <- progress T TS (ns-isval V)
             <- can-append-exp EL E APPEND.
```

However, this would be rejected by Twelf with an "output coverage error." The problem is that Twelf demands that the outputs of the inductive step be completely *unconstrained* - while it will allow case evaluation on the *input* to separate the different derivations of an input argument (`t-ref`, `t-int`, etc...), it does not currently do case evaluation on *outputs*. Both `prog-ref1` and `prog-ref2` put a constraint on the output of the inductive step, requiring it to be formed by `ns-steps` in the first case and `ns-isval` in the other, and Twelf cannot determine that all cases are covered between the two.

The solution is referred to as *factoring*. The output of the inductive step (the notstuckness of the subexpression) is passed to another temporary metalemma, which is here always simply called `lemma`, and *that* metalemma performs the case evaluation, which is now checking coverage on the input, as Twelf's current implementation demands.

```
lemma : notstuck S E -> notstuck S (ref E) -> type.
%mode lemma +NS1 -NS.
& : lemma (ns-steps E) (ns-steps (s-ref E)).
& : lemma (ns-isval V) (ns-steps (e-ref APPEND V))
     <- can-append-exp EL E APPEND.
%worlds () (lemma _ _). %total NS (lemma NS _).
prog-ref : progress (t-ref T) TS NS
            <- progress T TS NS1
            <- lemma NS1 NS.
```

Having introduced the idea of factoring, we can proceed to the case `t-add`. Notice the difference between this `lemma` and the one associated with case `t-ref`.

```
lemma : typed ST E1 int-tp -> notstuck S E1
     -> typed ST E2 int-tp -> notstuck S E2
     -> notstuck S (E1 + E2) -> type.
%mode lemma +T1 +NS1 +T2 +NS2 -NS.
& : lemma _ (ns-steps E) _ _ (ns-steps (s1-add E)).
& : lemma _ (ns-isval V) _ (ns-steps E) (ns-steps (s2-add E V)).
& : lemma t-int (ns-isval v-int) t-int (ns-isval v-int) (ns-steps (e-add SUM))
     <- can-sum N1 N2 SUM.
%worlds () (lemma _ _ _ _ _). %total [NS1 NS2] (lemma T1 NS1 T2 NS2 NS).
```

This `lemma` includes the fact that the two subexpressions have type `int-tp`, but then ignores this in the first two cases (which handle the first and second subexpressions taking a step). The third case is the case where both the subexpressions are values. The *canonical forms lemma* is what generally asserts that if something is a value with type `int-tp` then it is a natural number, but in most cases we can leave the canonical forms lemma implicit - Twelf itself checks our implicit assertion of canonical forms and in this case agrees that values with type `int-tp` must have the form `n N`, and therefore be valid inputs to `sum`.

However, the ability of Twelf to assert canonical forms in all cases would be akin to it being able to solve the general coverage checking problem, which is undecidable. In the few cases where Twelf did not believe a simple assertion of canonical forms, our experience was similar to the experience of other projects: "in cases of failure with spurious counterexamples it is in general possible to make the proof more explicit in such a way that it then passes the coverage checker" [13] by writing an explicit canonical forms lemma.

All other cases are similar. The `lemma` associated with case `t-bang` uses canonical forms and the `typed-projection` lemma discussed above, and in the case for `t-loc`, progress follows immediately from `v-loc`.

```
lemma : typed-store ST S
```

```
          -> typed ST E (ref-tp T)
          -> notstuck S E
          -> notstuck S (! E) -> type.
%mode lemma +ST +T +NS1 -NS.
& : lemma _ _ (ns-steps E) (ns-steps (s-bang E)).
& : lemma (&typed-store TL VL) (t-loc PROJ-T)
     (ns-isval v-loc) (ns-steps (e-bang PROJ-E))
     <- typed-projection TL PROJ-T PROJ-E _.
%worlds () (lemma _ _ _ _). %total NS1 (lemma ST T NS1 NS).
prog-bang : progress (t-bang T) TS NS
             <- progress T TS NS1
             <- lemma TS T NS1 NS.

prog-loc : progress (t-loc PROJ-T) TS (ns-isval v-loc).

lemma : {EF: exp -> exp} notstuck S E -> notstuck S (let E EF) -> type.
%mode lemma +EF +NS1 -NS.
& : lemma _ (ns-steps E) (ns-steps (s-let E)).
& : lemma _ (ns-isval V) (ns-steps (e-let V)).
%worlds () (lemma _ _ _). %total NS1 (lemma T NS1 NS).
prog-let : progress (t-let _ T) TS NS
            <- progress T TS NS1
            <- lemma EF NS1 NS.

%worlds () (progress _ _ _). %total T (progress T TS NS).
```

Note that there is no case for `t-var`. This is because we are only dealing with evaluating the different possible cases for the outermost step of the typing relation. The *only* way to create the `var-typed x T` judgement that is a requirement of `t-var` is as a dynamic clause introduced by `t-let`, and therefore `t-var` is not derivable at the top level. Twelf is capable of figuring this out without assistance, and so the case is omitted and the proof of progress is complete.

2.2.3 Weakening the store typing

Two lemmas which require induction over all the typing derivations are required for the progress theorem. The first is an assertion that adding to the storetype doesn't change the type of any expressions, formulated by Pierce as "If $\Gamma \mid \Sigma \vdash e : \tau$ and $\Sigma \subseteq \Sigma'$, then $\Gamma \mid \Sigma' \vdash e : \tau$" and formulated in Twelf like this:

```
weakening : typed ST E T
             -> subset-tp ST ST'
             -> typed ST' E T -> type.
%mode weakening +T +SUB -T'.
```

Most of the cases are extremely straightforward induction, and will be omitted because they are so similar to this one:

```
weak-sum : weakening (t-add T2 T1) S (t-add T2' T1')
            <- weakening T1 S T1'
            <- weakening T2 S T2'.
```

The one case that requires any kind of sophisticated reasoning is the case for `t-loc`, which needs a supporting lemma, `subset-projectable`, which asserts that `proj-tp` doesn't return a different result if the store typing is larger:

```
subset-projectable : proj-tp ST N T
                      -> subset-tp ST ST'
                      -> proj-tp ST' N T -> type.
%mode subset-projectable +P +S -P'.
& : subset-projectable proj-tp-z (subset-tp-s SUB) proj-tp-z.
& : subset-projectable (proj-tp-s P) (subset-tp-s SUB) (proj-tp-s P')
     <- subset-projectable P SUB P'.
%worlds () (subset-projectable _ _ _). %total P (subset-projectable P S P').

weak-loc : weakening (t-loc PROJ) S (t-loc PROJ')
    <- subset-projectable PROJ S PROJ'.
```

There is only one other interesting case, `t-let`, and that will be discussed in the next section. Once we have proven the `weakening` lemma, the same lemma for `typed-list`, which we also need, is simple to establish.

```
weakening-list : typed-list ST EL TL
                  -> subset-tp ST ST'
                  -> typed-list ST' EL TL -> type.
%mode weakening-list +T +SUB -T'.
weak-z : weakening-list tl-z S tl-z.
weak-s : weakening-list (tl-s TL T) S (tl-s TL' T')
          <- weakening T S T'
          <- weakening-list TL S TL'.
%worlds () (weakening-list _ _ _). %total T (weakening-list T S T').
```

2.2.4 Closed, open, and regular worlds

However, something very important happens at `t-let`:

```
weak-let : weakening (t-let F T) S (t-let F' T')
            <- weakening T S T'
            <- {x}{v: var-typed x Tp} weakening (F x v) S (F' x v).
```

Recall Section 2.1.7: If `t-let` is a recursive logic program, at `t-let` recursion passes over a λ-binding. When the recursive logic program `weakening` encounters `t-let`, it also has to pass over a λ-binding. It does this, as before, by introducing a new expression `x` and a new judgement `var-typed x Tp` into the context.

This was not an issue for the typing relation, as there was no attempt to establish the totality of the typing relation. Thus far, however, the coverage checker has been using the *closed worlds assumption*: the only derivations and cases that are possible are the ones that have already been declared. This is how Twelf knows that it won't suddenly run across a new natural number `r` that is neither `z` nor the successor of an existing natural, and that the rule $\Gamma \mid \Sigma \vdash \mathsf{loc}(l) : \mathsf{int}$ will not unexpectedly be introduced and break the progress theorem. Allowing for this sort of thing would be using the *open worlds assumption*, and it would clearly be disastrous for proving any metalemma; the assumption is "too general: it is impossible to guarantee that an induction principle, or the related recursive function covers all cases because the world is always subject to change" [12, Chapter 4]. However, the closed worlds assumption does not allow totality to be proven if the context is changed in the course of running a logic program, which is exactly what has to happen in order for `weak-let` to cross a λ-binding.

One of the contributions of Carsten Schürmann's Ph.D. thesis was to describe a way out of this problem, allowing the totality of the `weakening` metalemma to be established in a sound way. The induction described by the `weakening` metalemma will not allow the introduction of new natural numbers, or typing rules, or ways of deriving `sum`. It can, however, introduce two new things together as a *block*: a new expression `x` and the judgement `var-typed x T` for some (already derivable) type `T`. This utilizes the *regular worlds assumption*, and the regular world that was just described is encoded in this way:

```
%block var-tp : some {T:tp}
                 block {x:exp}{v:var-typed x T}.
```

Using this regular world description, the totality of the weakening metalemma can finally be established. Notice that the first set of parenthesis after the `%worlds` declaration now contains the block that was just defined.

```
%worlds (var-tp) (weakening _ _ _). %total T (weakening T S T').
```

2.2.5 Substitution

The other lemma requiring induction over the typing derivation that is required for the preservation lemma uses the same regular world defined for the weakening lemma. It is the substitution lemma, and is presented by Pierce as "If $\Gamma, x : \tau' \,|\, \Sigma \vdash e : \tau$ and $\Gamma \,|\, \Sigma \vdash e' : \tau'$, then $\Gamma \,|\, \Sigma \vdash [x \mapsto e']e : \tau$." The syntax of the encoding in Twelf is awkward because of the need to introduce variables multiple times, but the induction is relatively simple: the only important case is `sub-var`, which is responsible for replacing every instance where the type of the introduced variable was determined by the `t-var` derivation step with the derivation for $\Gamma \mid \Sigma \vdash e' : \tau'$. `t-int` and `t-loc` are combined into a single case, `sub-refl`, which also takes care of the typing of any expression where the introduced variable does not appear.

```
substitute : ({x} var-typed x Tp -> typed ST (EF x) T)
             -> typed ST E Tp -> typed ST (EF E) T -> type.
%mode substitute +F +T -T'.

sub-refl : substitute ([x][v: var-typed x Tp] T) T* T.
sub-sum : substitute ([x][v: var-typed x Tp] t-add (T2 x v) (T1 x v))
           T* (t-add T2' T1')
           <- substitute ([x][v: var-typed x Tp] T2 x v) T* T2'
           <- substitute ([x][v: var-typed x Tp] T1 x v) T* T1'.
sub-ref : substitute ([x][v: var-typed x Tp] t-ref (T x v)) T* (t-ref T')
           <- substitute ([x][v: var-typed x Tp] T x v) T* T'.
sub-bang : substitute ([x][v: var-typed x Tp] t-bang (T x v)) T* (t-bang T')
           <- substitute ([x][v: var-typed x Tp] T x v) T* T'.
sub-let : substitute ([x][v: var-typed x Tp] t-let (F x v) (T x v))
           T* (t-let F' T')
           <- substitute ([x][v: var-typed x Tp] T x v) T* T'
           <- {x'}{v': var-typed x' Tp2}
              substitute ([x][v: var-typed x Tp] F x v x' v') T* (F' x' v').
sub-var : substitute ([x][v: var-typed x Tp] t-var v) T* T*.

%worlds (var-tp) (substitute _ _ _). %total F (substitute F T T').
```

2.2.6 Preservation

The final step is proving the preservation theorem. Preservation is complicated in the presence of a storetype: it establishes that if e has type τ under the store typing Σ that also types the store μ, and if $\mu \mid e \longrightarrow \mu' \mid e'$, then under some store typing Σ' that is a superset of Σ and which also types

μ', e' has the same type τ. This translates into Twelf surprisingly cleanly, and actually appears quite similar to Pierce's declaration [11, Theorem 13.5.3].

```
preservation : typed ST E T
                -> typed-store ST S
                -> eval S E S' E'
                -> subset-tp ST ST'
                -> typed ST' E' T
                -> typed-store ST' S' -> type.
%mode preservation +T +TS +E -S -T' -TS'.
```

Two supporting lemmas are required. First, a trivial lemma that any store typing is a subset of itself:

```
subset-refl : {ST} subset-tp ST ST -> type.
%mode subset-refl +ST -S.
```

Second, a lemma which has an equally simple inductive proof, but which establishes an enormous number of things related to creating a new reference, and requires a little explanation. It involves one expression, e, which is a value and has type τ under the storetype Σ, and also a list of values e_{list}, which have the type τ_{list} under the same storetype. Given this introduction, if we derive e_{list} @ $e \Downarrow (e'_{\text{list}}, n)$, then it is always true that e'_{list} is still a list of values, and there exists a τ'_{list} with three properties. First, it is a superset of τ_{list}. Second, it types e'_{list} under the storetype Σ. Third, projecting the n^{th} element of τ'_{list} will return τ. The proof has only two simple cases, but they will still be described here due to the importance of the lemma. The proof proceeds by induction on the three derivations relating to e_{list}.

```
append-lemma : isval E
                -> typed ST E T
                -> isval-list EL
                -> typed-list ST EL TL
                -> append-exp EL E EL' N
                -> subset-tp TL TL'
                -> isval-list EL'
                -> typed-list ST EL' TL'
                -> proj-tp TL' N T -> type.
%mode append-lemma +V +T +VL +TL +AE -S -VL' -TL' -P.
& : append-lemma V T vl-z tl-z append-exp-z
     subset-tp-z (vl-s vl-z V) (tl-s tl-z T) proj-tp-z.
& : append-lemma V T (vl-s VL V*) (tl-s TL T*) (append-exp-s AE)
     (subset-tp-s S) (vl-s VL' V*) (tl-s TL' T*) (proj-tp-s P)
     <- append-lemma V T VL TL AE S VL' TL' P.
%worlds () (append-lemma _ _ _ _ _ _ _ _ _).
%total [VL TL AE] (append-lemma V T VL TL AE AT VL' TL' P).
```

The first cases we will consider are simple, the ones relating to the evaluation rules prefixed with an `s-`. Each case calls the inductive step on subexpressions that are evaluated, and calls `weakening` to make sure that typing derivations not related to the evaluation are updated with the new store typing.

```
pres-s1-add : preservation (t-add T2 T1) ST (s1-add E) S (t-add T2' T1') ST'
               <- preservation T1 ST E S T1' ST'
               <- weakening T2 S T2'.
```

```
pres-s2-add : preservation (t-add T2 T1) ST (s2-add E V) S (t-add T2' T1') ST'
               <- preservation T2 ST E S T2' ST'
               <- weakening T1 S T1'.
pres-s-ref : preservation (t-ref T) ST (s-ref E) S (t-ref T') ST'
              <- preservation T ST E S T' ST'.
pres-s-bang : preservation (t-bang T) ST (s-bang E) S (t-bang T') ST'
               <- preservation T ST E S T' ST'.
pres-s-let : preservation (t-let F T) ST (s-let E) S (t-let F' T') ST'
              <- preservation T ST E S T' ST'
              <- {x}{v} weakening (F x v) S (F' x v).
```

The proof of preservation is completed by considering the evaluation steps prefixed by `e-`. For the case `e-add`, preservation is immediate. For `e-ref` we call `append-lemma` to get the new store typing and the correct typing for the new location. However, `append-lemma` gives us $\Gamma \mid \Sigma \vdash \mu' : \Sigma'$, and so we need to call `weakening-list` on this judgement to get the correct store typing, which has to be with respect to Σ' rather than Σ. The case for `e-bang` is established by using the re-moded `typed-projection` lemma, the case for `e-let` from the substitution lemma, and as that covers all possible evaluation steps, the proof is complete.

```
pres-e-add : preservation (t-add t-int t-int) ST (e-add SUM) S t-int ST
              <- subset-refl _ S.
pres-e-ref : preservation (t-ref T)
              (&typed-store TL VL)
              (e-ref AE V) S (t-loc P)
              (&typed-store TL' VL')
              <- append-lemma V T VL TL AE S VL' TL'' P
              <- weakening-list TL'' S TL'.
pres-e-bang : preservation (t-bang (t-loc PROJ-T))
               (&typed-store TL VL) (e-bang PROJ-E)
               S T (&typed-store TL VL)
               <- typed-projection TL PROJ-T PROJ-E T
               <- subset-refl _ S.
pres-e-let : preservation (t-let F T') ST (e-let V) S T ST
              <- substitute F T' T
              <- subset-refl _ S.
%worlds () (preservation _ _ _ _ _ _). %total T (preservation T ST E S T' ST').
```

2.2.7 Safety

After having established progress and preservation, the proof of safety is straightforward. A program is safe if, being well typed under a store typing which also types the store, the program is not stuck after having run any number of steps. Preservation is used if the program steps, and progress is used if it does not.

```
safety : typed ST E T
          -> typed-store ST S
          -> run S E S' E'
          -> notstuck S' E' -> type.
%mode safety +T +TS +R -NS.
safe-end : safety T ST run-end NS
            <- progress T ST NS.
safe-step : safety T ST (run-step R E) NS
```

```
            <- preservation T ST E S T' ST'
            <- safety T' ST' R NS.
%worlds () (safety _ _ _ _). %total R (safety T TS R NS).
```

2.3 Error reporting in the type checker

Ignoring the safety metalemmas, which are not really intended to be run as logic programs, the language formalization has created two interesting logic programs: `run`, which simulates the dynamic semantics of the language, and `typed`, which is a type checker for the language.

However, as a type checker `typed` is not especially useful; it is not capable of reporting any information about why or where type checking failed, if it fails. Take the encoded example below, which is not well typed because the last line tries to add two expressions of type int and type int ref.

```
prog-bad =
let (ref (n (s (s z)))) [x]
(n z) + x.

%solve ty : typed nil-tp prog-bad T.
```

Obviously, `%solve` will be unable to determine a derivation for `typed nil-tp prog-bad T`, but it does not give the programmer any hints as to why that might have happened, merely reporting:

```
No solution to %solve found
%% ABORT %%
```

In order to consider how to deal with this problem, we will consider three ways in which the logic program `typed` could fail to come up with a type for a given expression: the lack of any typing rule which is able to cover an input expression (roughly equivalent to a match failure in ML), the inability for a non-recursive step of the logic program to establish a type, and the inability of a recursive step of `typed` to establish a type. After concluding that the last category is the only one we need to be concerned about, techniques for using Twelf's *tracing* abilities to report information in these cases will be considered.

2.3.1 Input coverage failure

The most obvious way for `typed` to fail is if it is asked to provide a type for an expression it cannot recognize. If a new kind of expression was introduced without any rules that could consider it, then the typing program would obviously be unable to establish a type for that expression. But recalling the discussion of regular worlds, only the defined expressions and expressions defined by let expressions can be introduced, and by examination it is clear that there will be a typing expression able to at least consider any possible expression that is encountered.

The discussion in this section will be informal, but it is worth noting that Twelf can formally establish that a logic program can accept any possible input, thereby avoiding this class of error, by defining the regular world for the logic program and then having Twelf check a `%covers` assertion, which is done for the Mini-ML type checker in the Twelf User's Guide [10, Section 9.2]. Doing this for the language presented here, however, would tell us very little, as in its current form `t-var` *alone* covers all possible input cases.

Free variables, of course, are one of the potential sources of type failure. Twelf's reaction in this case will be to simply fail upon any attempt to define a program with free variables. Attempting, for instance, to define this expression with a free variable `y` will cause Twelf to report an error "`Undeclared identifier y`"

```
prog-bad = let (n z) [x] x + y.
```

2.3.2 Typing failure in non-recursive cases

There are three "leaf" cases for the typing derivation, i.e. cases that do not rely on recursive calls to `typed`. The first of these, `t-int`, will always succeed if it is attempted, giving the expression type `int`. While `t-loc` does not always succeed, this is not an important consideration for error reporting; `loc`(l) is not a construct that should be available to the programmer, and all programs are initially checked with respect to the empty store typing [11, Section 13.3]. Finally we have `t-var`, and our (informal) understanding of the regular worlds condition for `typed` is that wherever a new variable `x` is introduced, the requirement for `t-var`, `var-typed x T` for some `T`, is introduced along with it, which means it, too, will always succeed. Therefore, the only non-recursive case that might potentially fail is `t-loc`, which the type checker does not need to consider, as it represents a construct unavailable to the programmer.

2.3.3 Typing failure in recursive cases

Of the recursive cases for `typed`, two of them cannot be the origin of any typing failures. There is no way that a typing error can originate at the level of `t-ref` or `t-let`. Assume the recursive calls (one in the case of `t-ref`, two in the case of `t-let`) succeed. The two predicates do not place any *constraints* on what happens to the output type, and therefore will in all cases succeed in returning a type if both their subgoals succeed.

There are only two cases left as a possible source of error, `t-bang` and `t-add`. These cases can be the origin of a type failure if the recursive calls do succeed in deriving a type but the single subexpression does not have the type τ `ref` for some τ in the first case, or if the two subexpressions do not have the type `int` in the second case.

Thus, the only two important sources of unreported error (as Twelf already reports all inappropriate free variables without any assistance) are where the subexpressions of $!e$ or $e_1 + e_2$ are typeable in general, but are *not* typeable given the constraints that `t-bang` and `t-add` place on their types.

2.3.4 Tracing failure

We can use this knowledge, along with the ability of Twelf to trace certain predicates during the execution of a logic program (described in the Twelf User's Guide [10, Section 11.5]), to build an error reporting mechanism into the type checker. We add a new logic program `fail` that can *never* succeed, and create new cases for typing addition and dereferencing that do not place any constraints on the types of the subexpression, but which require a derivation of the underivable `fail` in order to succeed.

```
fail-msg : type.
impossible : tp.

<Not_A_Ref> : fail-msg.
<Not_Both_Ints> : fail-msg.

fail : fail-msg -> type.
%mode fail +MSG.

f-add : typed ST (E1 + E2) impossible
        <- typed ST E1 T1
        <- typed ST E2 T2
        <- fail <Not_Both_Ints>.
f-bang : typed ST (! E) impossible
```

```
          <- typed ST E T
          <- fail <Not_A_Ref>.
```

As an example of how this would work, imagine that a typing derivation cannot be found for `t-add`. The logic program will search for other typing rules that are applicable to the same statement, and will encounter `f-add`. If it succeeds in finding typing derivations for `f-add` without any constraints placed on the outputs, it will then attempt to derive `fail <Not_Both_Ints>`, which will fail, causing `f-add` and the rest of the type checking to fail as well. However, by placing a trace on `fail`, the type checker outputs the message `fail <Not_Both_Ints>` from the type checker before failing.

The useful property of this method is that it is compatible with the safety proof - because Twelf realizes without assistance that `fail` cannot be satisfied, these error-correction cases can be added without changing any part of the safety proof.

2.3.5 Requiring determinism in the type checker

Unfortunately, the description in the last section does not actually describe what happens. Consider again our motivating example of trying to add an integer to a location:

```
prog-bad =
let (ref (n (s (s z)))) [x]
(n z) + x.

%solve ty : typed nil-tp prog-bad T.
```

The expectation is that this program will give the error `<Not_Both_Ints>`, but instead it gives *two* `fail` messages:

```
fail <Not_Both_Ints>
fail <Not_A_Ref>
```

The second fail message is the result of Twelf *backtracking*: after it fails to determine the correct type for the second argument to `+` (i.e. `y`), evaluation looks to see if there are any *other* derivations of the type of the *first* argument to `+` (i.e. `x`) - in fact, there is one, `f-bang`, and attempting to use it causes a trace message.

The solution is to tell Twelf to not backtrack when running `typed`, which can be done with the command `%deterministic typed.` This shouldn't make any difference in which expressions `typed` will construct a type for, and even if it does it does not threaten our safety property, a deterministc search will not produce any more typing derivations than a nondeterministc search, and an overly conservative type checker, while inconvienent, is not a threat to type safety.

This error reporting information, while more useful than no feedback at all, is still not extremely useful without also providing position information - the Fun compiler described in the next chapter will also deal with error reporting, and it *will* use positions in error messages.

2.3.6 The catch: output coverage

There is one last problem to which both this specification and the Fun specification are susceptible to some degree. It is a misconception of the way in which output constraints are handled. We have been making the implicit assumption when dealing with this rule:

```
t-add : typed ST (E1 + E2) int-tp
         <- typed ST E1 int-tp
         <- typed ST E2 int-tp.
```

that the two proof requirements operate by first typing E1 and E2 and then making sure that their type is int-tp. However, this is not exactly what is happening. In reality, the logic program is trying to see how it can *force* the integer type on the two expressions. The distinction is unimportant with respect to determining a type for E1, but it makes a difference in the context of the error reporting mechanism. In order to make type checking work correctly, it appears that the proof would need to be restructured to allow initial checks to be unconstrained, like this:

```
t-add : typed ST (E1 + E2) int-tp
        <- typed ST E1 T1
        <- typed ST E2 T2
        <- equals-tp T1 int-tp
        <- equals-tp T2 int-tp.
```

In the full Fun type checker, this appears to be a problem in two instances. The first instance is when the body of the program does not have the required type int. The second instance is a function body which does not have the type it is declared to have. In both of these situations the type checker may either report a spurious error in addition to the correct error or may alternatively present the wrong error altogether. In most cases, however, the type checker does present informative error messages, and more attention paid to this aspect of type checkers would surely have fruitful results. Real error reporting in the Twelf type checker, while not perfectly implemented in this project, at least appears extremely plausible.

Chapter 3

The verified Fun compiler

The previous chapter took a very close look at the techniques involved with writing a language formalization. This chapter takes a significant step back, looking at issues dealing with extending a small language like one presented in the last chapter into a somewhat larger language, in this case the language Fun, defined in Appendix A.

The first half of the chapter deals with scalability issues related to the formalization of larger languages. The second half of the chapter considers an interesting application of a language formalization - an integrated complier and simulator for the Fun specification which is built on top of the Twelf specification.

3.1 Goals of the Fun formalization

The goal of the Fun formalization was not merely to create a formalization of the Fun language, but to create within the logic system of LF a working model that would function as both a realistic type checker for the language and a full simulation of the language. The primary reason for this follows the spirit of Syme's research on Java type systems, where he worries that the process of checking that a formalization actually implements the language it claims to is both nontrivial and "a topic that has often been ignored by the theorem proving community" [16].

Syme's solution to this problem was to automatically generate a ML type checker and interpreter from his logical specifications in order to test the accuracy of his formalizations. That step is unnecessary here, as the specifications created in Twelf are already executable as logic programs. Syme notes that while his test programs are expressed as abstract syntax, "better would be the ability to parse, compile and run Java programs directly from the source code" [16]. As far as we know, this is the first project to either independently formulate or implement Syme's suggestion.

3.2 Developing the Fun formalization

Little has been said about the methodology of machine-checked theorem proving; Syme is an exception, presenting his methodology as a variant of the "waterfall" methodology used by software development [16]. His methodology has six steps:

1. Understand the Problem
2. Develop a Machine Acceptable Model
3. Validate the Model by Generating an Interpreter
4. Formulate All Key Properties

5. Sketch an Outline of the Proof

6. Convince the Machine

This project followed a very similar procedure, with two essential differences. One is that we generally attempted to prove things in Twelf and then discuss them in mathematical notation. This was a weakness of the project, and was only possible because most of the models were simple and relatively well-formalized elsewhere.

A good on-paper specification, along with a solid understanding of the reasons why the desired proofs are true, is absolutely crucial to a formalization project. One of the reasons why we switched from Tiger to Fun as a target language can be traced back to the initial decision to not create a rigorous specification of the *entire* Tiger language *before* beginning. Starting with a more rigorously defined on-paper understanding of Tiger might have been quite helpful; alternatively, if the initial approach had involved a better-defined language like Fun, the project might have been able to investigate more complex and interesting features, such as subtyping for Fun.

The other difference of emphasis between this project and Syme's is the centrality of an *modular* process to this project. The essence of this approach is captured by a design philosophy which we refer to as *feature-oriented formalization.*

3.2.1 Feature-oriented formalization

Robert Harper said in the context of a hand-checked formalization of Standard ML that "As a design tool, type theory gives substance to informal ideas such as 'orthogonality' and 'safety' and provides a framework for evaluating and comparing languages" [5]. This section will present these two informal ideas of *orthogonality* and *safety* not as purposes of type system formalization, but as a way of understanding two different organizing principles for language formalization.

Chapter 2 presented the language in the same way most small languages, and many larger ones, have been presented, for instance the well-studied Mini-ML example that is distributed along with Twelf as an example case study [10]. To review, the minimalist Fun formalization of Chapter 2 was presented in nine steps, four concerned with the encoding and five concerned with the safety proof:

- Syntax
- Dynamic semantics
- Static semantics
- Multi-step evaluation
- Progress
- Store type weakening
- Substitution
- Preservation
- Language safety

This is a very natural way to explain and present a language. It is focused around presenting steps towards the ultimate goals of the formalization, and as these goals usually involve type safety proofs, this can be considered a *safety-oriented* approach. The result of this design strategy is that while it is easy to prove new theorems about the language, it more difficult to add or remove capabilities from the language, as a new proof is one modular entity, but a new capability for the language requires modifying almost every module.

In practice this is not the most helpful way of building up a language formalization, though it may or may not be the best way to *present* a large formalization. The alternative presented here relies heavily on the logic programming metaphor of Twelf. In a safety-oriented presentation, each element can be considered to be "frozen" after its presentation - no more syntax is presented after the presentation of the dynamic semantics begins, for example. In fact, Twelf has a `%freeze` declaration that formalizes this informal concept, but is not used in this project.

This alternative formalization begins by specifying the types which make up the syntax (`exp`, for example), and then defines the structure of the partial functions which represent the semantics, and presents the form of the various metatheorems. Each capability or feature of the language is then presented along with the new syntax, static semantics, and dynamic semantics needed to encode that capability, and bundles this with the cases of each of the metalemmas pertaining to the capability. This approach can be thought of as *orthogonality-oriented* formalization after Harper's two categories, though *feature-oriented* is used here as a less tongue-twisting alternative. The full code for the feature-oriented Fun formalization is organized as follows:

- Setup for syntax, dynamic semantics, static semantics
- Define metatheorems
- Functions
- Tuples
- Integers
- Mutable store
- Let statements
- Control flow (if and while)
- Assert that all metatheorems are now total
- Language safety

This approach allows language features to be effectively inserted into or removed from the language while maintaining a proof that the smaller and larger language is type safe - removing control flow from this formalization, for instance, requires commenting out two lines in a configuration file, one containing the syntax and semantics, and the other containing the cases for the various lemmas requiring induction over the semantic relations. Because the relevant cases for the inductive proofs are bundled with the syntax and semantics, loading this new configuration still allows the safety proof to pass for the modified language!

This ability is useful in at least two capacities. First, it forces the formalizer to think about the dependencies existing between aspects of the language, as the alternative name, orthogonality-based formalization, suggests. Within the version of Fun used by this project, integers, mutable store, and control flow all depend on tuples, but only on one aspect, the unit value or empty tuple which is the return value of printint statements, assignments, and while loops. Furthermore, control flow is dependent on integers, as branching is conditional on the value of an integer. Allowing for these dependencies, all other language features are orthogonal and can be added and removed at will from the specification simply by editing configuration files.

A second advantage of feature-oriented formalization was observed in practice. When changes need to be made to certain aspects of the semantic relations or the safety proofs, the same incremental method used for formalization *design* proved itself useful as a method for formalization *maintenance*. It was easier, upon changing the definition of one of the safety proofs or one of the semantic relations, to "fix" the language with the new framework by steps, for instance, by first establishing safety for a language with tuples and nothing else. This method also allows for easy definition of features that did not appear in the full language formalization, for instance a debug feature that introduced only the unit value and unit type instead of requiring the full definition of tuples in order to have a unit type. A related benefit was that if there was a significant problem with a support lemma like substitution or storetype weakening, then the language features whose proofs relied on those lemmas could just be removed, and rather than removing the broken lemma in its entirety, the one-line assertion that the metalemma had been proven could be commented out until the proof was fixed.

For these reasons, compared to the common safety-oriented style, feature-oriented formalization allows for cleaner, more maintainable, and more extensible formalizations which make the orthogonality of different language features more explicit. There is a comparison to be drawn between the two organizational principles for formalization discussed here and aspect- and object-oriented programming; however, the terminology of safety- and feature-oriented formalization is less ambiguous, because either formalization approach could be thought of as "object-oriented" depending on whether one considers proofs or language features to be the "things" in a formalization.

3.3 Notes on the full Fun formalization

While most of the theoretically interesting aspects of the Fun formalization were covered in Chapter 2, a few properties of the formalization deserve mention here, especially the representation of primitive numbers, mutual recursion, input and output, and a few other issues related to efficient representation.

3.3.1 32-bit binary number representation

As mentioned in Chapter 2, the native Twelf versions of numbers are incompatible with the metatheorem capabilities of the language. The previous chapter implemented addition with successor naturals, but because the goal of the formalization was to allow for a faithful simulation of the language's dynamic semantics, the simple solution of successor naturals is unsatisfactory for two reasons. First, it does not allow for a faithful representation of the actual language because there is no concept of a negative number or of integer overflow error. Second, this approach can become extraordinarily inefficient when dealing with large numbers; Norrish noted that his use of HOL's successor naturals in a C formalization "slowed down many aspects of theorem-proving" [8, Section 6.3].

It probably would have been beyond the scope of this project to design and formalize a system of 32-bit integers and signed integer arithmetic, but luckily much of the work had already been done by the TALT project at Carnegie Mellon, which had a useful binary number representation as well as most of the operations that were needed to formalize Fun, though a few, such as multiplication, had to be added. The implementation is extraordinarily inefficient, and is far worse than successor naturals for small numbers, but it results in a far more faithful representation of the language with negative numbers and integer overflow, and unlike successor naturals it has performance that does not degrade substantially when dealing with very large numbers.

It is an important caveat that there are none but the most informal claims that this implementation of binary arithmetic, especially in the case of binary multiplication, is "correct" in any sense. Doing so formally would require attaching some sort of semantic meaning to the binary numbers and operations. The TALT project appeared to show correctness through a comparison to natural numbers; in the presence of overflow error and signed arithmetic this would be quite difficult.

3.3.2 Mutual recursion

Fun also includes a capacity for mutual recursion, and treats functions as values. However, it is generally quite problematic in higher order logic to have mutual recursion. A general approach to dealing with this problem was one of the persistent issues with the Tiger formalization; Fun allows for a specialized solution because all functions are defined at compile-time.

The reason that this solution is possible is that Fun does not allow nested functions in order to avoid the need for passing around closures (a strategy also used by C-like languages), and so a function value is essentially just a code pointer. The formalization reflects this idea of code pointers, representing a function value as a natural number which acts as an index into a list of functions, just as locations are indexes into a list of expressions. Making this work elegantly requires a little magic with the parser, which will be discussed below in Section 3.4.3.

The typing relation is also extended with a function typing that works in exactly the same way the store typing works. The list of functions is added to the program state, though unlike other elements of the program state, the function table never changes throughout execution of the program.

3.3.3 Input and output

The Fun language also includes a capability for input and output. Input is done by way of a list of binary numbers that is added to the program state. If the list is empty, then `readint` will always return the binary representation of 0 in order to keep the program state from being stuck. Two other options for out-of-input behavior are throwing an exception and going into an infinite loop, but exceptions require lots of additional evaluation cases, and going into an infinite loop, while type safe, is not a practical option for a real language.

Output is simulated by the trace mechanism discussed in Section 2.3.4. The trace is put on a `print` predicate which always succeeds (as the progress theorem requires), as opposed to `fail` which always fails (as the progress theorem also requires).

3.3.4 A few hacks: list reuse and operator typing

It is also worth noting that tuples and tuple types are represented with the same terminology used to represent the storetype - a list of expressions and a list of types. This representation strategy requires the addition of a logic program `eval-list` to complement the `typed-list` already presented; furthermore, it effectively means that all of the inductive definitions must include an annoying and simple extra case dealing with lists, and the inductions dealing with the expression typing relation and with the list typing relation must be performed simultaneously, which luckily can be represented quite simply in Twelf.

The reuse of lists serves to eliminate a lot of repetition of uninteresting parts of the formalization proof, and a second strategy removes a lot of uninteresting repetition in the proof of progress. Recall from Chapter 2 that almost every case of the proof of progress requires a factoring lemma, which makes it less clear where the interesting parts of the proof of progress actually happens. The larger specification utilizes an operator-based strategy to simplify the proof of progress. There are two types of operators, binary and unary, each of which can have polymorphic type, meaning that everything from addition to assignment can be represented as an operator. Note that this does *not* mean that the language is polymorphic, simply that some of its built-in operators are polymorphic - the unary `ref` operator, for instance, which has type τ `ref` where τ is the type of its subexpression. Due to this representation strategy, $!x$ is encoded as `(op1 bang x)` rather than as `(! x)` as it was in Chapter 2, $a + b$ is encoded as `(op2 addop a b)`.

One factoring lemma is then defined for *all* unary operators, another for *all* binary operators, and the final case where all arguments are values is handled by a support lemma, `can-op1` for unary operations and `can-op2` for binary operations, which includes as inputs the fact that both inputs are values of the expected type. This is all specified in the initial setup of the language framework, so all that is required of the progress proofs for the individual features of the language is that they establish a single case of the relevant `can-op` lemma, essentially showing that if two values have the correct type, the operator can always be applied to those values.

3.4 Compiling into LF

Once the LF specification was somewhat stable, we developed a compiler that took Fun input and compiled it into the LF representation of the program's abstract syntax. The LF abstract syntax is created from the abstract syntax tree produced by a standard compiler frontend, though it could be created directly within the parser. This section describes a couple of issues with that compilation process.

3.4.1 Defined constants

In order to use the TALT-style binary number representation, there must be some way of translating the ML integer 9 into the Twelf object `(one $ zero $ zero $ one ... nil$)`. Rather than performing this within ML, two logic programs were written that translated back and forth between binary number representations and internal Twelf 32-bit numbers.

Just as there is no formal argument for the correctness of the binary operators discussed above in Section 3.3.1, there is no particular reason to expect the translation process to work correctly. It could fail by producing the wrong result, or by not terminating at all, but because this translation happens outside of the realm of language safety, this does not threaten the safety proof - these "untrusted" elements are most accurately considered part of the parser which just happen to be written in Twelf, and in this project the parser must be trusted (or rather, it does not concern itself with the question of trusting the parser).

Thus, the first element of the file containing the compiled abstract syntax is a series of declarations which use the Twelf `%define` command to bind the results of running logic programs as constants.

```
%define #14 = N
%solve _ : word->binary n32 positive 14 N.
```

Because the parser ensures that every constant will be defined in this header section, it can represent the integer n in the abstract syntax as just #n.

The choice between doing the translation in ML or in Twelf is relatively unimportant; however, the strategy of using defined constants has a certain advantage over doing the translation "in place." First, it makes the code produced much more compact and readable, as the objects representing 32-bit binary numbers are large and obviously are not as easily readable by humans as decimal numbers are. Second, Twelf only "looks inside" definitions when it has to, and so if a number that has not been modified by any of the arithmatic operators gets printed out with `printint`, it will be printed as, say, `#3` instead of as the object `(one $ one $ zero ... nil$)`.

3.4.2 Identifiers

Another issue is making sure that identifiers are placed in a different name space. The compiled version of $3+5$ in the LF abstract syntax for Fun is `(op2 addop #3 #5)`. However, consider a naive encoding of the perfectly valid Fun program (let addop=3 in addop + 5):

```
let #3 [addop]
op2 addop addop #5
```

This would be rejected by Twelf as poorly typed, because the first `addop`, which is supposed to have the type `oper` in Twelf, has been shadowed by the `addop` which has type `exp`.

The solution is to make sure all identifiers are prefixed by a prefix such as `id-`, making the above example perfectly acceptable to Twelf. A similar strategy could allow for multiple name spaces by using different prefixes.

3.4.3 Making mutual recursion work

Consider what happens when the parser encounters a variable. The parser only knows enough to prefix it with `id-`, but it might be bound by a `let` statement, or it might be bound because is one of the functions. If it is the latter case, then we must make sure that that variable will not be a free variable, keeping in mind that we may be working with an expression that is in one of the functions or an expression that is in the body of the program.

The LF Fun compiler deals with this by doing a conversion on the entire program. Functions are all bound as a block at the beginning of the file, and so as the parser encounters them it creates two lists. One is a list of functions (the "function table"), and the other is a list of binders which associates each function identifier with a "function pointer," the natural number representing the function's index in the function table.

3.4.4 Example conversion

The following example is a valid Fun program, which is useful for demonstrating most of the issues discussed in the previous sections.

```
fun f(x: int) : int = let f = 4 in x + f
fun g(x: int) : int = f x
  in
let x = 7 in
5 + f (g x)
```

Figure 3.1 approximately represents the complied version of this program, with the one difference being that this representation has none of the position information that will be considered in Section 3.5.2. The figure demonstrates the defined constants and the division of the single program into a set of function binders, a function table, and the body of the program.

3.4.5 Shadowing

One final issue deals with the connection between scope and mutual recursion. Twelf *cannot* look inside of lambda bindings, and this is in some ways one of the major weaknesses of Twelf - Twelf has no way whatsoever of distinguishing between `(let #1 [x] let #2 [x] x + x)` and `(let #1 [y] let #2 [x] x + x)`, for instance, even though a shadowing is happening in the first instance that is not happening in the second instance. The ability to do this is the only way that Twelf would be able to enforce a requirement that two mutually recursive functions do not have the same name. Therefore, if enforcement of such a requirement is desired, it can *only* be enforced in the parser. This is not terribly problematic; because the parser is creating a list of function labels anyway, the list module could simply make a check for duplication upon insertion, abstracting that check away from the parser.

Typechecking and simulation aren't even done on the program until the function identifiers are substituted, and so this process cannot lead to a type safety violation - if it is done wrong, it may result in programs meaning something different than what the programmer intended, however. If the parser does not enforce the no-duplication rule - and ours does not - then if multiple functions are declared with the same name, the last one to be declared is the *only* one that is in scope *anywhere* - including in the bodies of all the other functions with the same name.

3.5 Type checking and simulation

If the compiled LF representation of the program loads successfully in Twelf, then it is certain that there are no free variables, and the type checker can be run.

```
% -----Defined constants----- %
%define #4 = P
%solve _ : word->binary n32 positive 4 P.

%define #5 = P
%solve _ : word->binary n32 positive 5 P.

%define #7 = P
%solve _ : word->binary n32 positive 7 P.

program_all : prog =
% -----Function labels----- %
lam_prog z [id_f]
lam_prog s(z) [id_g]
end_prog (
% -----Function table----- %
fun inttp inttp ([id_x]
(let (int #4)
       [id_f]
(op2 addop id_x id_f)))
    $fundecl
fun inttp inttp ([id_x]
(op2 app id_f id_x))
    $fundecl
 nil_fundecl)
% -----Program body----- %
(let (int #7)
       [id_x]
(op2 addop (int #5) (op2 app id_f (op2 app id_g id_x)))).
```

Figure 3.1: Twelf representation of the Fun program from Section 3.4.4

The compiler controls Twelf by loading the configurations in different files - for instance, after loading the abstract syntax tree, it loads a logic program that replaces all of the variables representing functions with the correct pointer into the function table, which then allows the type checker to be run.

3.5.1 The type checker

Type checking is done by a file which performs two checks. First, it checks that the function table has the type it claims to have by calling the logic program `funs-typed`, which in turn calls `typed` on the body of each function. Second, it runs the `typed` logic program on the body of the program to make sure that it is well typed and has type `int`.

The same operator abstraction that was discussed in 3.3.4 reduces the number of failure conditions that must be addressed significantly, as there can essentially be only one failure check covering all binary operators and one covering all unary operations. The operator typing also results in leaving the output of a recursive call initially unconstrained, a good thing from an error reporting perspective as discussed in Section 2.3.6

3.5.2 Position reporting

A significant extension to the error reporting discussed in Section 2.3 is that the Fun specification allows reporting of the *position* at which an error occurred.

Normally this would require a lemma that if an expression has a type at one position, it has that type at any position, which would require an annoying induction over the entire typing derivation. The way around this is to have only *one position*, which is then simply *redefined* with the name of each individual position that is needed (positions have to be collected by the parser in the same way that defined constants are). Because Twelf never looks inside of definitions if it doesn't have to, the failure tracing reports the correct renaming of the single position whenever it has to output error information, and because there is only one position, only a few entirely trivial lemmas need to be added.

3.5.3 Simulation

A process similar to the one for typechecking allows simulation of program execution. The input integer list is encoded in the same way that the defined constants are. Because the type checker requires that the body of the program have type `int`, the final result of running the program, if the program terminates, will be a 32-bit binary number object. There is another logic program, which simply must be trusted in the same way that the program that turns integers into 32-bit binary number objects must be trusted, that runs the encoding in the other direction and returns the final value of the program: a sign (`positive` or `negative`) and an integer.

3.6 The promise of formal type checkers

This chapter has gone a significant distance past Chapter 2, which was primarily concerned with the details of how to use Twelf do things like encode a language, prove properties of the language, and provide rudimentary typechecking with error reporting. This chapter dealt with two topics relating to writing a compiler that type checked and evaluated the Fun language from Appendix A using Twelf.

The first consideration was related to scaling the small project into a larger one. Primitive operations, efficiency and code reuse considerations, and the organizational strategy of feature-oriented formalization were all discussed as part of this framework. The second part of the chapter showed how a frontend compiler could create the abstract syntax tree of a language in Twelf, and could then interface with Twelf to do the work of type checking and/or simulating execution of the language.

The important aspect of the compiler was not the interface between ML and Twelf, however, and the chapter's organization in some ways served to deemphasize in many ways the most important thing that was happening. The second half of the chapter was the design of a compiler that could represent its abstract syntax trees as LF terms and could represent its static and dynamic semantics as functions in the recursive function space M_2^+.

Having considered the second half of the chapter, the first discussed a safety proof which reasons about the *code of the compiler*. By writing terms in a certain way, and by writing the type checker and simulator of the compiler in a certain way, something incredible has been gained - powerful tools allowing the language designer to reason about the compiler. The type checker for this compiler is verified, because it is the executable specification of the language's dynamic semantics. Read backward, this chapter begins with two parts of the compiler being designed in a specialized way which allows the syntax and semantics to be reasoned about, a point which we will return to in Section 4.2.6.

Chapter 4

Conclusion

We have shown that Twelf is a adequate tool for formalizing programming languages, and that it is accessible for developers outside of the Twelf development community. Chapter 2 discussed fine-grained details of the language formalization process. Chapter 3 stepped back to discuss aspects of full-language formalizations, also considering how a Twelf-like system might be able to serve as the basis for a type checker which could be reasoned about with formal methods. This chapter takes yet another step back. Twelf is currently the only language that supports this kind of project; the majority of this chapter discusses Twelf's strengths and weaknesses for the task we have used it for. We conclude by examining future topics of study suggested by this project.

4.1 Notes on proving metatheorems in Twelf

One of the series of computer checked formalizations of subsets of Java which were developed in the second half of the last decade utilized a custom-made theorem prover DECLARE. The report on the Java formalization developed by Syme, who was also DECLARE's author, included a set of eight factors for evaluating the suitability mechanized provers [16]. Twelf satisfies or comes close to satisfying all of these criteria, though after discussing the eight criteria individually we will include a few more important criteria.

4.1.1 Syme's eight criteria

What underlying logic is used, and what is its expressiveness?

The elegance of higher-order abstract syntax in the Twelf framework was enormously important to the success of this project. The metatheorem prover is complicated and, at least in its current form, its proofs should probably not be taken without a grain of salt, especially in light of significant soundness violations in Twelf 1.4 and remaining issues in Twelf 1.5 relating to the constraint domains [15, Constraint Domains]. As for the expressiveness of Twelf's proving abilities, there are very important language properties that have not yet shown themselves to be easily formulated or provable in Twelf, but the metatheory of M_2^+ is definitely a sufficiently expressive framework for formulating and proving the kinds of type safety properties considered by this project, even if it is not sufficient for other tasks, such as understanding closure conversion.

How much automated reasoning support is provided to avoid tedious reasoning?

We have already discussed our reasons for not utilizing Twelf's theorem prover; however, the coverage checker is also an automated reasoning process with a great deal of power, as show by its frequent ability to establish canonical forms lemmas on its own. It had a few annoying quirks, such as the lack of support for output factoring, but in almost all cases it was sufficient; the error messages produced by the coverage checker, which fall in this category as well, were extremely helpful.

What language is used for specifications, and can specifications be written in a natural style?

As Chapter 2 showed, the syntax and semantics of the language can be defined almost as succinctly as a rigorous mathematical treatment of the same syntax and semantics. More complicated syntax than the types of syntax presented in Chapter 2 did present expressive problems, however. Lists were annoying as they needed to be redefined for each new thing we needed a list of, but the "fake polymorphism" offered by our system of text search/replace/save was not incredibly difficult to manage.

Several features of the Tiger system that were also experimented with as part of this project, such as mutual recursion without a function table and mutually recursive record types, required either a great deal of extra notation or a group of complex "extraction" functions to deal with appropriately. The answer to the question of what is the "best" approach or set of approaches to mutual recursion in higher-order abstract syntax is one of the primary unfulfilled goals of this project.

How are proofs expressed, and can arguments be formulated succinctly and naturally?

This point should be considered with the last one; the beauty of the LF's dependent type system is that a single notation is used for the encoding of the language's syntax, the specification of its semantics, and the expression of proofs about the language. The Twelf notation is an extremely concise way of representing derivation trees; consider as an alternative Pierce's discussion of algorithmic subtyping [11, Section 16.2], which requires a much more expansive notation to describe its manipulations of derivation trees.

What assistance is given the construction of proofs?

It is significant that Syme presents this point as distinct from the second point about automating away tedious language features. The Twelf theorem prover is designed to provide proof realizations, i.e. total logic functions equivalent to the metatheorems which this paper used; however, as mentioned in Section 1.2.3, this feature is disabled at present.

We believe there is probably an enormous potential in a mixed strategy of automatic proofs being incorporated into proofs of the style of this project, and once the theorem prover is fixed so that the two approaches can be incorporated, it would be interesting to see an annotated proof of the style of this report's Chapter 2 describing a proof using a hybrid strategy.

How does the tool support the maintenance of specifications and proofs under incremental changes?

This was the great advantage of the feature-oriented proving approach - as long as a set of features were grouped together in the same file, as most similar groups of features were, they could be added and removed from the language along with the cases relating to them in the various inductive proofs simply by commenting out two lines in a configuration file. The development of this technique was a major reason for the success of the Fun formalization.

The project also experimented with a more significant incremental change, successfully modifying the framework created for the Fun formalization to create a formalization of a language using l-values in the style of Tiger. The relative ease of this was a significant testament to the ability of feature-oriented designs in Twelf to be adapted to support very different languages.

Are the documents produced readable? Can they be validated by researches unfamiliar with the tool?

The second part of Chapter 2 is a case study in using the Twelf code itself as an explanation of an inductive proof; this is an approach which we believe is successful here and which certainly holds great promise. The meaning of Twelf code is not "self evident," but with sufficient annotation, like the annotation provided in Chapter 2, it appears capable of standing on its own as a representation of an inductive proof.

Does the tool support exploratory modes of work?

The ability to test loading individual declarations and to rapidly reload entire logic systems through the Emacs interface gives Twelf excellent exploratory capacities. One feature which would make it significantly easier to use the metatheorem prover in an exploratory way is an equivalent of the "hole lemma" - the ability to unsafely make assertions which can later be returned to and proven to make the proof safe again.

Twelf does have an "unsafe mode" which allows unproven assertions to be made within the theorem prover, but the ability to assert unproven metatheorems for proofs using the totality checker, not the theorem prover, is not a capability which is currently implemented. Sarkar describes a way to cause Twelf to believe the totality of metalemmas which have not been proven total [15, Holes in Metalemmas], but this trick halts the loading of configuration files and would be entirely impractical to use in proofs with many holes. A more elegant extension of the technique already available to the Twelf theorem prover would be an extremely useful in helping to prevent the annoying problem of realizing that one has gone through a great deal of trouble constructing a tedious supporting lemma for what turns out to be an incorrectly formulated metatheorem. The `%trustme` declaration slated for future inclusion in Twelf is hopefully designed to address this problem.

4.1.2 Three additional criteria

Twelf clearly performs satisfactorily with respect to all of Syme's criteria; however, we would like to suggest that Syme's criteria are incomplete. If computer-verified formalization methods are to be more widely adopted, it is crucial that each formalization not be expected to develop its own theorem proving environment. Syme's criteria, while they deal adequately with the internal capabilities of a theorem-proving system, do not adequately touch on questions of accessibility of the development environment to outsiders; this is understandable, as Syme followed the observed trend of being a developer of his own system.

As mentioned in the introduction, we believe that one of the main contributions of this project is that it was probably the first of its scale to utilize the metatheorem capabilities of Twelf *without* both a significant background in LF and access to the developers of the Twelf system. Our three additional criteria all deal with aspects of a question that has been central in many ways to this project, "How accessible is the theorem proving environment to outsiders?"

How thorough and useful is the documentation available?

The Twelf User's guide is a useful document, but it is limited in several ways; the most important is that is more focused on describing features than with teaching users strategies and approaches for solving common problems, a problem especially apparent with the documentation of the coverage checker. In short, The Twelf User's Guide is not intended to be a teaching document. It is our hope that The Twelf Wiki [15] will be able to fill the need for a community teaching resource, becoming a collection of what contributor Todd Wilson referred to as "tutorial articles," which is a shortcoming of the previously available documentation.

How easy is it to decipher errors having to do with limitations of the theorem proving environment?

The coverage checker is necessarily an incomplete solution to a problem (as the general problem is undecidable), and therefore it will refuse to confirm certain facts which may seem obvious to the user. One of the most annoying aspects of dealing with a theorem proving system is understanding the kinds of errors that are generated when the user attempts something that is impossible, but for reasons that are completely non-obvious to the user; in terms of AUTOMATH-style uses of Twelf, this frustration is primary in terms of certain "strictness" requirements, about which Appel declares, "For most users (including me!) this is the most mysterious part of Twelf" [2, Part 6].

In the context of Twelf metatheory, there are at least three mysterious aspects. First is output factoring, which is confusing the first couple of times it is encountered, but which gets easier when the idea that outputs must be totally unconstrained is drilled into the user's head. The second issue is when coverage checking fails on a "spurious" case, such as when Twelf refuses to automatically assert a canonical forms lemma that is, in fact, true. This, like the strictness requirements, is a problem that improves with time and experience with the system and the types of error messages that appear. There were other more mysterious errors encountered when using the coverage checker, such as instances where adding cases to the substitution metalemma would cause the coverage checking for seemingly unrelated parts of the proof to stop working; we are still unsure whether some of these are bugs in Twelf or are like strictness requirements, i.e. problems that are reasonable because of the necessary incompleteness of the coverage checker, but which we simply don't understand. In all of these cases, the continued development of the Twelf system, along with more user documentation, will make these sorts of problems easier to deal with.

How long does it take to become an effective user of the system?

This is the million-dollar question. Familiarity with Twelf's syntax and operations in a different context certainly helped this project; we were able to formalize interesting facts in a matter of a few weeks, though there was obviously much that we still had to learn at that point.

In any case, the learning curve doesn't appear to be harsher than the "couple of months" estimate for comparable systems such as HOL or ACL2 [6]; moreover, because of the extraordinary similarity between the representations of standard inductive proofs and Twelf representations of those proofs, we believe that the the learning curve may possibly be be much lower for Twelf metatheory than for comparable systems. This project, in our view, represents a good indication what is currently achievable in a year by a single theorist outside of the Twelf developer community starting with minimal knowledge of the Twelf metatheory. However, we hope that this experience can make the smallest of contributions towards "paving the road" for future projects, allowing the next project to go a little farther at a slightly faster speed.

4.2 Further work

This project suggests an enormous number of other projects, and this final section considers a small number of these projects.

4.2.1 Modifying Fun's function table

It would be quite interesting to see what would happen if the Fun formalization could be extended in type safe way to allow the function table to be modified during program execution. Even more interesting might be investigating whether the function table could use this ability to implement recursion with backpatching.

4.2.2 Approaches to mutual recursion

Function backpatching is only one way to implement recursion. The various formalizations of fragments of Tiger relied heavily on determining more general forms of mutual recursion, and this was achieved with only limited success. There is a lot of work still to be done on finding the most effective solutions to the problem of recursion in higher-order abstract syntax.

4.2.3 Extending l-values

We completed a specification of a language that used l-values, that is to say a language where locations weren't values and where a new reference was generated at every `let` statement, as opposed to Fun where references have to be explicitly declared and dereferenced. However, there were some tricky problems resulting from the way l-values were set up which made it quite difficult to have more than one kind of l-value (arrays, for example, introduce another kind of l-value). It would be interesting to investigate what would have to be changed to allow different kinds of l-values to be introduced in this safe language.

Another topic also deals with the l-value specification. Java and other languages have the property that the value of a variable representing an l-value does not have to be provided at the time of the l-value's definition, but the type checker will complain if the program might attempt to use it before it is initialized. It would be quite interesting to try and translate the reasoning of such a system into Twelf.

4.2.4 Investigating error reporting

Section 2.3.6 presented some issues with error reporting in the type checker as it currently stands. It would be interesting to further develop this error reporting mechanism to solve the problems with it as they currently stand. Even more significant, however, might be the consideration of other ways in which Twelf itself could be modified to allow for more error reporting in a more elegant way than the way we implemented it, which is after all a bit of a hack.

4.2.5 Subtyping for Fun

Of the features of the original Fun language, the only one left out of our Fun formalization is subtyping; it would be a fairly large project, but it would be an interesting and significant feat to add subtyping to the formalized Fun. Formalizing Fun with subtyping would probably be done best by proving language safety with respect to a declarative typing relation with subsumption, and then providing an algorithmic typing relation and proving its adequacy as a metatheorem.

4.2.6 ML-Lex, ML-YACC... ML-Type?

A much larger project might involve a recurring suggestion in this project, using a Twelf-like notation to create a general language for type checkers. M_2^+ is very much a general language that can specify the abilities that most type checkers need; is it possible to create another way of representing M_2^+ specifications that can be more easily integrated into a compiler without sacrificing the ability to use metalogic to reason about them?

Bibliography

[1] A. Appel and A. Felty. Dependent types ensure partial correctness of theorem provers, 2002.

[2] Andrew W. Appel. Hints on proving theorems in Twelf. www.cs.princeton.edu/~appel/twelf-tutorial, 2000.

[3] Bryan Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge.

[4] Karl Crary and Susmit Sarkar. A metalogical approach to foundational certified code, 2003.

[5] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[6] Matt Kaufmann and J Strother Moore. A flying tour of ACL2.

[7] Tobias Nipkow and David von Oheimb. Java-light is type-safe — definitely. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 161–170, San Diego, California, 1998. ACM Press.

[8] Michael Norrish. C formalised in HOL.

[9] Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[10] Frank Pfenning and Carsten Schürmann. Twelf user's guide 1.4, 2002.

[11] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[12] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, October 2000.

[13] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF, 2003.

[14] Rob Simmons. A new system for arithmetic. Internal documentation for the Princeton FPCC project, July 2005.

[15] Rob Simmons et al. The Twelf Wiki: http://fp.logosphere.cs.cmu.edu/twelf.

[16] Don Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[17] David Walker. The definition of Fun. Available at:
http://www.cs.princeton.edu/courses/archive/spring05/cos320/assignments/fundef.htm.

[18] David Walker. Course lectures for COS 320: Compilers, 2005.

[19] Daniel Wang. Higher-order theories for first-order syntax. Draft, intended for submission to ICFP 2005, April 2005.

Appendix A

The Definition of Fun

Fun is a small language designed for teaching compilers. It belongs to the Algol family of C-like languages because it avoids the need for closures by disallowing nesting of functions but still allowing functions to be values, as opposed to languages like Tiger which are in the Pascal family of languages which achieve this by not allowing functions to be values. This definition of Fun is adapted from the definition of David Walkers's Fun language which was created as a teaching language for a compliers course [17] - the language itself is fundamentally just Walker's Fun without subtyping and with integer inputs.

A.1 Syntax

This describes the concrete syntax of Fun, though the abstract syntax is extremely similar with a few caveats.

Integers:	i	$\in$	$\{0, 1, 2, 3, ...1073741823\}$
Identifiers:	x	$\in$	$\{\mathsf{a-z}, \mathsf{A-Z}\}\{\mathsf{a-z}, \mathsf{A-Z}, \mathsf{0-9}, \_\}^*$, except reserved words
Unary Operators:	un	$::=$	$\sim \mid \mathsf{printint} \mid \,!\, \mid \#i \mid \mathsf{not}$
Binary Operators:	bn	$::=$	$+ \mid - \mid \times \mid = \mid < \mid := \mid \& \mid \mathsf{\|\|} \mid ;$
Types:	τ	$::=$	$(\tau) \mid \mathsf{int} \mid \{\tau, ..., \tau\} \mid \tau \to \tau \mid \tau\ \mathsf{ref}$
Expressions:	e	$::=$	$(e) \mid i \mid x \mid un\ e \mid e\ bn\ e \mid \{e, ..., e\} \mid e\ e \mid \mathsf{readint}$
			$\mid \mathsf{ref}(e : \tau) \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{while}\ e\ \mathsf{do}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$
Function declarations:	f	$::=$	$\mathsf{fun}\ x\ (x{:}\tau) : \tau = e$
Program:	p	$::=$	$f...f\ \mathsf{in}\ e$

A.1.1 Notes on the concrete syntax

- Comments may appear between any two tokens. Comments start with /*, end with */, and may be nested.
- White space (the newline character, carriage returns, tabs, and spaces) may appear between any two tokens.
- Fun reserves the following keywords:

  fun, in, let, while, do, if, then, else, printint, readint, ref, not, int

- For types, ref has higher precedence than $\rightarrow$, which is right associative. For expressions, binary operators are left associative with the exception of the ; operator which is right associative. Precedence is as follows (highest precedence first):

  1 - Unary operators (except for not)

  2 - Function application

  3 - $\times$

  4 - $+$, $-$

  5 - $=$, $<$

  6 - not

  7 - &, ||

  8 - ;

A.1.2 Notes on the abstract syntax

- The abstract syntax has an additional piece of syntax, locations (l), which are introduced with two additional expressions, loc l and fn l, which are not available to the programmer.
- The current implementation of the verified Fun compiler does not allow projection of more than 36 elements from a tuple.
- In the abstract syntax, integers are 32-bit signed binary numbers.
- The concrete syntax does not treat ref as a unary operator or function application ad a binary operator, but the abstract syntax does.
- Several operators are only abbreviations for other expressions. The underscore in the last shorthand simply designates that the identifier used must be outside of the name space of other variables.

  a & $b \Longrightarrow$ if a then b else 0

  a || $b \Longrightarrow$ if a then 1 else b

  not $a \Longrightarrow$ if a then 0 else 1

  $\sim e \Longrightarrow 0 - e$

  a ; $b \Longrightarrow$ let _ = a in b

A.2 Tables

There are several different kinds of tables - the function table ν and its typing Υ, and the store μ and its typing Σ. Tables are accessed using locations.

The formalization treats locations as natural numbers and tables as lists, but here locations will simply be presented as uninterpreted indexes to tables. Tables allow three operations, though only the first will be used on ν and Υ.

- $\mu(l) \Downarrow e$ - lookup the item at the given location, assuming it exists
- $\mu \hookleftarrow e \Downarrow (\mu'\ ,\ l)$ - add an element at a fresh location (returns the modified table and the new location)
- $\mu[l] := e \Downarrow \mu'$ - update an element at a known location, assuming it exists (returns the modified table)

A.3 The program transformation

A single program must be reinterpreted as a function table and an expression to be understood by the static and dynamic semantics. This can be done by initially considering all of the variables not bound by a let or by the fact that they are the argument to a function to be free variables. In order for the program to be well-formed, every free variable must match a function identifier.

The function table is built by placing all the functions in the function table and replacing every free variable x with fn l where l is the location of the function named x. This replacement must be done in the bodies of all the functions in the function table and in the body of the program; if the same identifier is used for two functions, the one declared last is the one used for this substitution.

A.4 Static semantics

A.4.1 Program typing

First, for each element fun $x_f\ (x{:}\tau_1) : \tau_2 = e$ in ν at location l, Υ is populated with $\tau_1 \rightarrow \tau_2$ at location l.

The function table is then well-formed if for each fun $x_f\ (x{:}\tau_1) : \tau_2 = e$ in ν, it is true that for the *empty* context Γ and the *empty* store typing Σ, $(\Gamma[x \mapsto \tau_1] \mid \Sigma \mid \Upsilon \vdash e : \tau_2)$ can be derived.

The body of the program is required to have type int.

A.4.2 The typing relation

The type of all programs is initially checked under the empty context and the empty store.

$$\frac{}{\Gamma \mid \Sigma \mid \Upsilon \vdash i : \mathsf{int}}\ \texttt{t-int} \qquad \frac{}{\Gamma[x \mapsto \tau] \mid \Sigma \mid \Upsilon \vdash x : \tau}\ \texttt{t-var} \qquad \frac{}{\Gamma \mid \Sigma \mid \Upsilon \vdash \mathsf{readint} : \mathsf{int}}\ \texttt{t-readint}$$

$$\frac{\Gamma \mid \Sigma \mid \Upsilon \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \mid \Sigma \mid \Upsilon \vdash e_2 : \tau}{\Gamma \mid \Sigma \mid \Upsilon \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \texttt{t-let}$$

$$\frac{\Gamma \mid \Sigma \mid \Upsilon \vdash e : \mathsf{int} \quad \Gamma \mid \Sigma \mid \Upsilon \vdash e_1 : \tau \quad \Gamma \mid \Sigma \mid \Upsilon \vdash e_2 : \tau}{\Gamma \mid \Sigma \mid \Upsilon \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}\ \texttt{t-if}$$

$$\frac{\Gamma \mid \Sigma \mid \Upsilon \vdash e_1 : \mathsf{int} \quad \Gamma \mid \Sigma \mid \Upsilon \vdash e_2 : \{\}}{\Gamma \mid \Sigma \mid \Upsilon \vdash \mathsf{while}\ e_1\ \mathsf{do}\ e_2 : \{\}}\ \texttt{t-while} \qquad \frac{\Gamma \mid \Sigma \mid \Upsilon \vdash e_1 : \tau_1 \quad \mathsf{optype}(un) = \tau_1 \rightarrow \tau}{\Gamma \mid \Sigma \mid \Upsilon \vdash un\ e_2 : \tau}\ \texttt{t-op1}$$

$$\frac{\Gamma \mid \Sigma \mid \Upsilon \vdash e_1 : \tau_1 \quad \Gamma \mid \Sigma \mid \Upsilon \vdash e_2 : \tau_2 \quad \mathsf{optype}(bn) = \tau_1, \tau_2 \rightarrow \tau}{\Gamma \mid \Sigma \mid \Upsilon \vdash e_1\ bn\ e_2 : \tau}\ \texttt{t-op2}$$

$$\frac{\Gamma \mid \Sigma \mid \Upsilon \vdash e_1 : \tau_1 \quad ... \quad \Gamma \mid \Sigma \mid \Upsilon \vdash e_n : \tau_n}{\Gamma \mid \Sigma \mid \Upsilon \vdash \{e_1, ..., e_n\} : \{\tau_1, ..., \tau_n\}}\ \texttt{t-tuple}$$

$$\frac{\Upsilon(l) \Downarrow \tau}{\Gamma \mid \Sigma \mid \Upsilon \vdash \mathsf{fn}\ l : \tau}\ \texttt{t-fn} \qquad \frac{\Sigma(l) \Downarrow \tau}{\Gamma \mid \Sigma \mid \Upsilon \vdash \mathsf{loc}\ l : \tau}\ \texttt{t-loc}$$

A.4.3 Operator typing

The optype function is defined by the following table:

$+$	$\mathsf{int}, \mathsf{int} \Rightarrow \mathsf{int}$
$-$	$\mathsf{int}, \mathsf{int} \Rightarrow \mathsf{int}$
$\times$	$\mathsf{int}, \mathsf{int} \Rightarrow \mathsf{int}$
$=$	$\mathsf{int}, \mathsf{int} \Rightarrow \mathsf{int}$
$<$	$\mathsf{int}, \mathsf{int} \Rightarrow \mathsf{int}$
$:=$	$\tau\ \mathsf{ref}, \tau \Rightarrow \{\}$
Function application	$\tau_1 \rightarrow \tau_2, \tau_1 \Rightarrow \tau_2$
$\sim$	$\mathsf{int} \Rightarrow \mathsf{int}$
printint	$\mathsf{int} \Rightarrow \{\}$
!	$\tau\ \mathsf{ref} \Rightarrow \tau$
$\#i$	$\{\tau_1, ...\tau_i, ...\tau_n\} \Rightarrow \tau_i$
ref with type annotation τ	$\tau \Rightarrow \tau\ \mathsf{ref}$

A.5 Dynamic semantics

The dynamic semantics have a concept of a value, represented by identifiers labeled v_1, v_2, v_3...

$$\text{Values:} \quad v \quad ::= \quad i \ \mid \ \{v, ..., v\} \ \mid \ \mathsf{loc}\ l \ \mid \ \mathsf{fn}$$

The program state is a 4-tuple consisting of a function table ν, which does not change during execution of the program, the store μ, a list of integers representing the input i_l, and an expression e.

Operators

$$\frac{\nu \mid \mu \mid i_l \mid e \longrightarrow \nu \mid \mu' \mid i_l \mid e'}{\nu \mid \mu \mid i_l \mid un\ e \longrightarrow \nu \mid \mu' \mid i_l \mid un\ e'}\ \texttt{s-op1}$$

$$\frac{\nu \mid \mu \mid i_l \mid e_1 \longrightarrow \nu \mid \mu' \mid i_l \mid e_1'}{\nu \mid \mu \mid i_l \mid e_1\ bn\ e_2 \longrightarrow \nu \mid \mu' \mid i_l \mid e_1'\ bn\ e_2}\ \texttt{s1-op2} \qquad \frac{\nu \mid \mu \mid i_l \mid e \longrightarrow \nu \mid \mu' \mid i_l \mid e'}{\nu \mid \mu \mid i_l \mid v\ bn\ e \longrightarrow \nu \mid \mu' \mid i_l \mid v\ bn\ e}\ \texttt{s1-op2}$$

Functions

$$\frac{\nu(l) \Downarrow \mathsf{fun}\ f\ (x{:}\tau)\ :\ \tau = e}{\nu \mid \mu \mid i_l \mid (\mathsf{fn}\ l)v \longrightarrow \nu \mid \mu \mid i_l \mid [x \mapsto v]e}\ \texttt{e-fn}$$

Tuples

$$\frac{\nu \mid \mu \mid i_l \mid e_i \longrightarrow \nu \mid \mu' \mid i_l \mid e_i'}{\nu \mid \mu \mid i_l \mid \{v_1, ..., v_{i-1}, e_i, ..., e_n\} \longrightarrow \nu \mid \mu' \mid i_l \mid \{v_1, ..., v_{i-1}, e_i', ..., e_n\}}\ \texttt{e-tuple}$$

$$\frac{}{\nu \mid \mu \mid i_l \mid \{v_1, ..., v_i, ..., v_n\} \longrightarrow \nu \mid \mu' \mid i_l \mid v_i}\ \texttt{e-proj}$$

Integers

We are not specifying the operation of the arithmatic operators (which implement 32-bit binary arithmetic with overflow), and the function most of these operators is to throw evaluation to one of the primitive operations which we are not discussing. Three exception are the rules dealing with I/O. The rule for printint steps to an empty tuple without considering I/O side effect of printing out the integer, and the rules for readint either pull the first element out of the input list or return 0 if no more elements exist.

$$\frac{}{\nu \mid \mu \mid i_l \mid \mathsf{printint}\ i \longrightarrow \nu \mid \mu \mid i_l \mid \{\}}\ \texttt{e-printint} \qquad \frac{i_1 + i_2 \Downarrow i_3}{\nu \mid \mu \mid i_l \mid i_1 + i_2 \longrightarrow \nu \mid \mu \mid i_l \mid i_3}\ \texttt{e-add}$$

$$\frac{i_1 \times i_2 \Downarrow i_3}{\nu \mid \mu \mid i_l \mid i_1 \times i_2 \longrightarrow \nu \mid \mu \mid i_l \mid i_3}\ \texttt{e-mul} \qquad \frac{i_1 - i_2 \Downarrow i_3}{\nu \mid \mu \mid i_l \mid i_1 - i_2 \longrightarrow \nu \mid \mu \mid i_l \mid i_3}\ \texttt{e-sub}$$

$$\frac{i_1 = i_2 \Downarrow i_3}{\nu \mid \mu \mid i_l \mid i_1 = i_2 \longrightarrow \nu \mid \mu \mid i_l \mid i_3}\ \texttt{e-eq} \qquad \frac{i_1 < i_2 \Downarrow i_3}{\nu \mid \mu \mid i_l \mid i_1 < i_2 \longrightarrow \nu \mid \mu \mid i_l \mid i_3}\ \texttt{e-lt}$$

$$\frac{}{\nu \mid \mu \mid i :: i_l \mid \mathsf{readint} \longrightarrow \nu \mid \mu \mid i_l \mid i}\ \texttt{e-readint-s}$$

$$\frac{}{\nu \mid \mu \mid \mathsf{nil} \mid \mathsf{readint} \longrightarrow \nu \mid \mu \mid \mathsf{nil} \mid 0}\ \texttt{e-readint-z}$$

Mutable state

$$\frac{\mu\ \texttt{@}\ v \Downarrow (l\ ,\ \mu')}{\nu \mid \mu \mid i_l \mid \mathsf{ref}(v : \tau) \longrightarrow \nu \mid \mu' \mid i_l \mid \mathsf{loc}\ l}\ \texttt{e-ref}$$

$$\frac{\mu(l) \Downarrow e}{\nu \mid \mu \mid i_l \mid !\ \mathsf{loc}\ l \longrightarrow \nu \mid \mu \mid i_l \mid e}\ \texttt{e-bang} \qquad \frac{\mu[l] := v \Downarrow \mu'}{\nu \mid \mu \mid i_l \mid \mathsf{loc}\ l := v \longrightarrow \nu \mid \mu' \mid i_l \mid \{\}}\ \texttt{s-bang}$$

Variable binding and control flow

$$\frac{\nu \mid \mu \mid i_l \mid e_1 \longrightarrow \nu \mid \mu' \mid i_l \mid e_1'}{\nu \mid \mu \mid i_l \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \longrightarrow \nu \mid \mu' \mid i_l \mid \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2}\ \texttt{s-let}$$

$$\frac{}{\nu \mid \mu \mid i_l \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow \nu \mid \mu' \mid i_l \mid [x \mapsto v]e}\ \texttt{e-let}$$

$$\frac{\nu \mid \mu \mid i_l \mid e \longrightarrow \nu \mid \mu' \mid i_l \mid e'}{\nu \mid \mu \mid i_l \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \nu \mid \mu' \mid i_l \mid \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}\ \texttt{s-if}$$

$$\frac{e = e_1 \text{ if } i = 0,\ e = e_2 \text{ otherwise}}{\nu \mid \mu \mid i_l \mid \mathsf{if}\ i\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \nu \mid \mu' \mid i_l \mid e}\ \texttt{e-if}$$

A.6 A few sample programs

A.6.1 tuple.fun - Definition of tuples

Prints the first and second input, returns 4.

```
fun fst (x:{int,int}):int = #0 x
fun snd (x:{int,int}):int = #1 x
fun print (x:int):{} = printint x

  in

let x = {readint,readint} in
print (fst x); print (snd x); 4
```

A.6.2 fact.fun - Factorial

Prints the first input, then computes the factorial of the input.

```
fun fact(x : int) : int =
 if (x < 0 || x = 0)  then 1 else x * fact (x - 1)

 in

let x = readint in
printint x;
fact x
```

A.6.3 funv.fun - Functions as values, left-associative function application

Stores the first two inputs in `a` and `b`, respectively. If `a` is 0, returns $2 \times (\mathtt{b} + 10)$, otherwise returns $2 \times (\mathtt{b} - 10)$.

```
fun f (x:int):int->int = if (x = 0) then g else h
fun g (x:int):int = x + 10
fun h (x:int):int = x - 10

  in

let a = readint in
let b = readint in
let afun = f a in
afun b + f a b
```

A.6.4 evenodd.fun - Mutual recursion

Returns 2 if the first input is even, 1 if it is odd. Loops forever if the first input is negative.

```
fun iseven (x:int):int =
  if x = 0 then 1 else isodd (x - 1)
fun isodd (x:int):int =
  if x = 0 then 0 else iseven (x - 1)

  in

let a = readint in
if iseven a then 2 else 1
```