A Library Based Approach to the Verification of Languages with Linear Types

Never Stand Still | Faculty of Engineering | School of Computer Science and Engineering

Michael Alexander Sproul

Supervisor: Ben Lippmeier

Foundations for Low-level Software Verification

Computer systems form an integral part of modern society, both in the form of personal devices and critical infrastructure. Ensuring the correct operation of computer hardware and software is therefore unarguably a worthwhile endeavour. One emerging technique for the construction of robust software systems is the use of mathematical formalisations and proofs of correctness. In this paradigm, desirable properties of the software can be proven true using a computer-based *proof assistant,* which itself relies on a minimal amount of trusted code. To construct proofs about code written in a given programming language, the language's *semantics* must be specified mathematically. Unfortunately for the would-be software verifier, most popular programming languages lack formal semantics and are therefore not amenable to verification techniques. The C Programming Language, in which large amounts of low-level systems software is written, is an example of a language that is difficult to reason about because of its murky semantics. For my thesis I focussed on the formal semantics of languages with strong type systems that are hopefully more well-suited to low-level software verification than C.

Type Soundness

Type systems for programming languages are said to be sound if well-typed programs are guaranteed not to get stuck when evaluated according to the language's operational semantics. I proved soundness for a small linearly-typed language based on DILL, using the Coq theorem prover. In the process, I created a library for **context splitting** that might be of use in other language formalisations.

Dual Intuitionistic Linear Logic (DILL)

Dual Intuitionistic Linear Logic is a logical system that ensures **linear** assumptions are used only once. By the Curry-Howard correspondence, DILL's logic corresponds to a typed programming language, in which linear values are only used once. In the context of systems software, we can use variants of linearity to enable destructive updates for uniquely referenced values, or automatic memory management without garbage collection, as in Mozilla's new Rust language.

We write a *typing judgement* $\Gamma; \Delta \vdash t : A$ to signify that the term (or expression) t has the type A under the environments $\Gamma; \Delta$, which record the types of free variables in t. To determine which typing judgements are valid, we use deduction rules as shown in Figure 1. If the premises (judgements above the line) are true, then we can deduce the conclusion (the judgement below the line). Figure 2 shows a typing derivation for a function that duplicates its input argument x. This is possible because the input argument is of a duplicable type, denoted $!A$. In general, duplicable variables get stored in the first environment Γ, while linear ones get stored in Δ.

Context Splitting

DILL's product type $A \otimes B$ represents a pair of values and is inhabited by terms of the form $t \otimes u$ if $t : A$ and $u : B$, as demonstrated by its introduction rule $(\otimes -I)$. In order to guarantee that linear variables are used exactly once in $t \otimes u$ it is required that the linear context (Δ_1, Δ_2) is the result of joining the two variable-disjoint linear contexts of t and u. Alternately, we can view this as the environment for the pair being *split* into the two environments for t and u. This is the **context splitting** operation that is at the core of many substructural type systems, and is the focus of our Coq library.

The library I've developed includes a definition of context splitting (Figure 4), and a collection of lemmas about its properties. For example, splitting a context is associative and commutative, and we can state these properties as lemmas. The most difficult to prove lemmas were the ones concerned with the insertion of new variables and types, which arose during the proof of soundness for the linear lambda calculus. Figure 3 shows a graphical demonstration of context splitting for a string concatenation function. Each split represents an application of the $(\multimap -E)$ rule for function applications.

Figure 1: Typing Rules for Dual Intuitionistic Linear Logic

$$\frac{}{\Gamma, x : A; \varnothing \vdash x : A}\ \text{(Int-Var)} \qquad \frac{}{\Gamma; x : A \vdash x : A}\ \text{(Lin-Var)}$$

$$\frac{}{\Gamma; \varnothing \vdash * : \mathbb{I}}\ \text{(Unit-I)} \qquad \frac{\Gamma; \Delta_1 \vdash t : \mathbb{I} \quad \Gamma; \Delta_2 \vdash u : A}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let } * \textbf{ be } t \textbf{ in } u : A}\ \text{(Unit-E)}$$

$$\frac{\Gamma; \Delta_1 \vdash t : A \quad \Gamma; \Delta_2 \vdash u : B}{\Gamma; \Delta_1, \Delta_2 \vdash t \otimes u : A \otimes B}\ (\otimes\text{-I}) \qquad \frac{\Gamma; \Delta_1 \vdash u : A \otimes B \quad \Gamma; \Delta_2, x : A, y : B \vdash t : C}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let } x \otimes y : A \otimes B \textbf{ be } u \textbf{ in } t : C}\ (\otimes\text{-E})$$

$$\frac{\Gamma; \Delta, x : A \vdash t : B}{\Gamma; \Delta \vdash (\lambda x : A.t) : A \multimap B}\ (\multimap\text{-I}) \qquad \frac{\Gamma; \Delta_1 \vdash u : A \multimap B \quad \Gamma; \Delta_2 \vdash t : A}{\Gamma; \Delta_1, \Delta_2 \vdash (u\ t) : B}\ (\multimap\text{-E})$$

$$\frac{\Gamma; \varnothing \vdash t : A}{\Gamma; \varnothing \vdash\ !t : !A}\ (!\text{-I}) \qquad \frac{\Gamma; \Delta_1 \vdash u : !A \quad \Gamma, x : A; \Delta_2 \vdash t : B}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let } !x : A \textbf{ be } u \textbf{ in } t : B}\ (!\text{-E})$$

Figure 2: Typing Derivation for a Duplication Function in DILL

$$\dfrac{\dfrac{}{\varnothing; x : !A \vdash x : !A}\ \text{(Lin-Var)} \quad \dfrac{\dfrac{\dfrac{}{y : A; \varnothing \vdash y : A}\ \text{(Int-Var)}}{y : A; \varnothing \vdash\ !y : !A}\ (!\text{-I}) \quad \dfrac{}{y : A; \varnothing \vdash y : A}\ \text{(Int-Var)}}{y : A; \varnothing \vdash\ !y : !A}\ (!\text{-I})}{y : A; \varnothing \vdash (!y \otimes !y) : (!A \otimes !A)}\ (\otimes\text{-I})}{\varnothing; x : !A \vdash \text{let } !y : A \text{ be } x \text{ in } (!y \otimes !y) : (!A \otimes !A)}\ (!\text{-E})$$

$$\frac{\varnothing; x : !A \vdash \text{let } !y : A \text{ be } x \text{ in } (!y \otimes !y) : (!A \otimes !A)}{\varnothing; \varnothing \vdash (\lambda x : !A.\ \text{let } !y : A \text{ be } x \text{ in } (!y \otimes !y)) : !A \multimap (!A \otimes !A)}\ (\multimap\text{-I})$$

Figure 3: Context Splitting for a String Concatenation Function

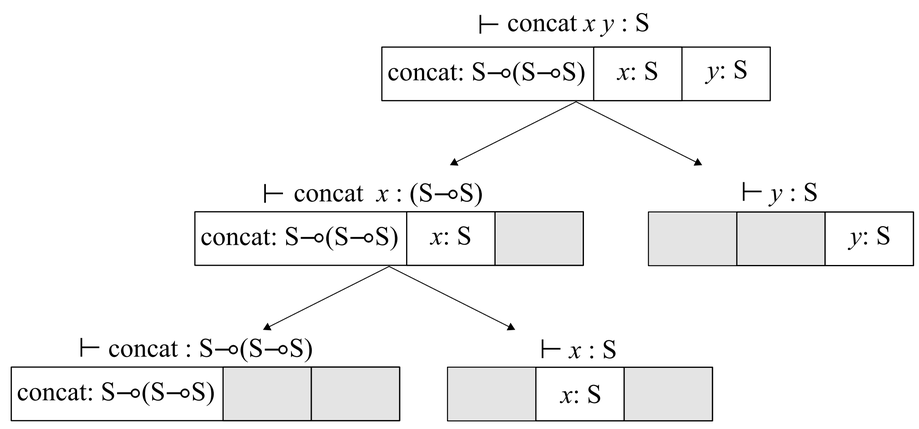

Figure 4: Coq Code for an Inductive Definition of Context Splitting

```
Inductive split_single {A} : option A -> option A -> option A -> Prop :=
  | split_none : split_single None None None
  | split_left (v : A) : split_single (Some v) (Some v) None
  | split_right (v : A) : split_single (Some v) None (Some v).

Inductive context_split {A} : env A -> env A -> env A -> Prop :=
  | split_nil : context_split nil nil nil
  | split_cons E E1 E2 v v1 v2
      (SplitElem : split_single v v1 v2)
      (SplitPre : context_split E E1 E2) :
      context_split (v :: E) (v1 :: E1) (v2 :: E2).
```

Conclusions

Although it was initially hoped that the context splitting library would be useful in the formalisation of more advanced languages, this is probably unrealistic. The first reason for this is that more advanced languages tend to involve type system extensions that obsolete context splitting as we have defined it, or contain so many other features that context splitting only makes up a small fraction of the overall proof effort. Secondly, the approach to context splitting whereby unavailable values are entirely erased is subsumed by an approach based on separation algebras whereby the number of available copies of each variable is tracked in the typing context. This approach is used to great effect by François Pottier for SSPHS, and Conor McBride for a fusion of linear and dependent typing.

Despite the only moderate successes of my library-based approach to context splitting, the future for substructural languages looks bright. Mozilla's Rust programming language represents an opportunity for substructural typing to break into the mainstream, and it might not be long before tools exist to allow the construction of next-generation verified operating systems, networking software and desktop environments.