2 Language Design

I am currently working on integrating refinement types into Zélus, a synchronous programming language designed for modeling hybrid systems by combining differential equations with discrete jumps. Our ultimate goal is to leverage these refined types to verify hybrid system programs against their specifications during compilation. This capability has the potential to significantly reduce costs and save lives when applied to real-world systems, such as autonomous aerial vehicles. However, translating specifications into refinement types is often challenging for programmers. In some cases, the system may reach undesirable states because the programmer failed to correctly define variable subtypes. As a result, the compiler might incorrectly verify the system as safe. As Hoare emphasized, a programming language should be designed to make errors difficult to introduce and easy to diagnose, rather than allowing them to propagate undetected. This principle underscores the importance of security in language design, which I agree how crucial it is. To address these challenges, we might consider synthesizing refinement types directly from specifications. By automating this process, programmer errors can be reduced and we can enhance the reliability of hybrid system verification during compilation.

At the same time, I have encountered a practical limitation of Hoare's hint on simplicity. In the synchronous domain, certain specialized features, like clock refinement or temporal contracts, prove indispensable for describing time-sensitive behaviors. While these additions do raise the complexity of the language, they also make it far more expressive for embedded systems or reactive applications that require guaranteed real-time responses.

In conclusion, the process of integrating refinement types into Zélus shows the balance between simplicity, robustness, and expressiveness in programming language design. Automating refinement type synthesis could reduce errors and improve hybrid system verification, enhancing safety in critical applications. While Hoare's principle emphasizes simplicity, some features are necessary for modeling time-sensitive behaviors, demonstrating the trade-off between complexity and practical utility in designing reliable systems.

3 Simple Operational Semantics

To extend the arithmetic expressions (Aexp) of the IMP language with a division operator(/), we first need to extend the Abstract Syntax.

$$\begin{aligned} e ::=\, & n \text{ for } n \in \mathbb{Z} \\ & \mid x \text{ for } x \in \text{L} \\ & \mid e_1 + e_2 \text{ for } e_1, e_2 \in \text{Aexp} \\ & \mid e_1 - e_2 \text{ for } e_1, e_2 \in \text{Aexp} \\ & \mid e_1 * e_2 \text{ for } e_1, e_2 \in \text{Aexp} \\ & \mid e_1 / e_2 \text{ for } e_1, e_2 \in \text{Aexp} \end{aligned}$$

We keep the Bexp and Com as the same. And keeping all the evaluation rules we can add the evaluation rules for division operator. We should take care of division by zero error as well. The second rule below specify that if the denominator of the division is zero, the division is undefined thus by adding division we make our system incomplete.

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow n_2 \quad n_2 \neq 0}{\langle e_1/e_2, \sigma\rangle \Downarrow n_1 \div n_2}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow n_1 \quad \langle e_2, \sigma\rangle \Downarrow 0}{\langle e_1/e_2, \sigma\rangle \text{ is undefined}}$$

Here $\div$ denotes the integer division.

4 Language Feature Design, Large Step

Let's extend the natural-style operational semantics judgment $\langle c, \sigma\rangle \Downarrow \sigma'$ with one new rule for dealing with the let command.

$$\frac{\langle e, \sigma\rangle \Downarrow n \quad \langle c, \sigma[x := n]\rangle \Downarrow \sigma'}{\langle \text{let } x = e \text{ in } c, \sigma\rangle \Downarrow \sigma'[x := \sigma(x)]}$$

Here we create a temporary state $\sigma[x := n]$ where x is bound to n and perform other commands c based on that information. And σ' denotes the state after evaluating c under the temporary state. After we finish evaluating c we restore the original value of x by $\sigma'[x := \sigma(x)]$ but we keep the other changed values in σ' as is.

5 Language Feature Design, Small Step

Let's first extend the set of redexes,

$$\begin{aligned} r ::= & x \quad x \in L \\ & \mid n_1 + n_2 \\ & \mid x := n \\ & \mid \text{skip;c} \\ & \mid \text{if true then } c_1 \text{ else } c_2 \\ & \mid \text{if false then } c_1 \text{ else } c_2 \\ & \mid \text{while b do c} \\ & \mid \text{let } x = n \text{ in c} \end{aligned}$$

Here, n represents an evaluated arithmetic expression. And we can extend the reduction rules by adding the following rule,

$$\langle \text{let } x = n \text{ in c}, \sigma\rangle \to \langle \text{c}; x := \sigma(x), \sigma[x := n]\rangle$$

Here, we substitute the variable x and reduce the body c within the new scope. After executing c we add an assignment x to recover the old value of it. And finally we can extend the contexts by adding the following contexts to H,

$$H ::= \ldots \mid \text{let } x = H \text{ in } c$$

This allows reduction to occur in the evaluation of the bound expression e.