Declarative and Algorithmic Implementation of Refinement Types

F. C. Slothouber

Delft University of Technology
June 28, 2019

Abstract

Type systems and their accompanying checkers provide support for the programmer to write better and safer code, faster, with less effort and with less errors. There are however properties that can not be checked at compile time yet. Refinement types are potentially the solution. They can prove properties of the behaviour of code without actually running and therefore avoid costly bugs in software. This is done by decorating types with predicates that tell something about the value of that type. This paper discusses an implementation of a refinement type system for a functional language.

Keywords – Refinement Types, Declarative Type System & Algorithmic Type Checker

I. Introduction

Type systems and their accompanying checkers provide support for the programmer to write better and safer code, faster, with less effort and with less errors. Take for instance the following invalid line of code: `5 + "12"`. Conventional type checkers will fail this line before run-time and give some form of feedback as to why the line of code is considered invalid. In this particular case it will indicate that addition can only be done with numerical values, like 5, but not with string representations of numbers, like "12". If "12" is to be added to 5 then it should be parsed to a number first. Altering the line results the code snippet to: `5 + parse("12")` and the type-checker will successfully run and the code can be executed.

The aforementioned example illustrated the use and benefit of types in programming languages during programming. The type-checker prevented a bug in the code without potentially catastrophic failures if the bug would have been deployed. It is reasonable to assert that if type-systems and their checkers would become more advanced, then more errors could be prevented. Take for instance the arguably faulty implementation of the mathematical division function in the next code snippet.

```
fun div(a: int, b: int) → int = a / b
```

What would happen if b is equal to zero? Division by zero is undefined behaviour and this could cause an exception to be thrown or a similar construct. Remedying this faulty behaviour could be done in multiple different ways. To start, exception handling could be done, this is however not a particular elegant solution. If functions ultimately become more complex and consequently start throwing a multitude of different exceptions then the code handling the errors becomes unclear and actual functional code becomes less apparent.

An improvement would be that the function would never be called with input that is not supported. This could be done by doing a check in the form of an if-statement in the code before calling a function. It is not favorable to do this check all the time. Performance is unnecessarily reduced, code complexity is increased and the human programmers could make mistakes, forgetting to do the checks or check the wrong properties.

Refinement types potentially provide the solution. Y. Mandelbaum, D. Walker, and R. Harper [1] state that *"One of the major goals of programming language design is to allow programmers to express and enforce properties of the execution behavior of programs."*. At the moment, conventional type-systems are unable to enforce many properties which are apparent at compile time [1].

Refinement types helps expressing and enforcing these properties. As the name suggest, refinement types allow ordinary types to be refined. Normally this means that a type can be written with an accompanying predicate [2] [3], a predicate that tells something about the value of the type. For example: $int\ \{\ v > 0\ \}$ indicates the positive integers, v is special variable indicating the value of the type itself [2]. Then $int\ \{\ v < 0\ \}$ indicates

the negative integers and *int { 4 < v ∧ v < 9 }* indicates all the integers between 4 and 9. Types are therefore interpreted as sets of values and a refinement type is the subset of the values of the base type such that for every value in the subset the predicate holds [4].

To illustrate the use case in a more practical fashion, take a look at the divisor in the earlier example. The divisor can be any integer but the value zero, so the parameter b could be refined with the predicate $v \neq 0$.

```
fun div(a: int, b: int { v ≠ 0 })
    → int = a / b
```

A refinement-type-checker checks and enforces that *div* is never called in a program with an argument for b that interprets to zero. If the program is validated, then the program can be safely executed and no *DivisionByZeroException* or a similar construct is ever thrown while not checking all the time if b is not equal to zero. This will save performance, reduce code complexity and avoid bugs.

This paper researches the definition of refinement-type-systems, the implementation of refinement-type-checkers and the use of refinement types. In this paper a formal definition of a refinement type system for a functional programming language is defined, an implementation of that type system is developed and examples illustrating the use cases are given.

The main objective is to *implement a fully featured type checker that validates refinement types*, for the source check the GitHub repository [5]. Contributions made in the paper are:

- An introduction to refinement types, examples with regard to the application of refinement types and difference compared to conventional type systems.
- Declarative definition of a refinement type system for a functional programming language.
- Insights in the practical and algorithmic implementation of a type checker of such a type system.

II. Syntax and Semantics

The syntax for the language is given in 1. The programming language is functional and deliberately kept simple. The language is functional because that programming paradigm lends itself for relatively simple implementation. This does not necessarily mean that refinement types cannot be implemented for imperative, object-oriented or other programming paradigms. For instance a refinement-type-system has been developed for Typescript [6]. Typescript is a multi-paradigm programming language. It is including functional, imperative and object-oriented. The language is kept simple because more advanced features can be desugared to simple features and because other features do not automatically imply an impact on the type system. This way the type system and the typing rules can be kept concise, simple and less repetitive.

A *program* is defined as a collection of global *functions*. There are no other constructs like global variables and importing of other files.

Each function has an identifier, potential parameters, which are explicitly typed, a return type and an expression.

All possible expressions are kept to a minimum. The expressiveness is large enough to give some insightful examples that illustrate the potential use of refinements and small enough to avoid repetitiveness in the definition later on.

The language supports the four main arithmetic operations: *addition*, *subtraction*, *multiplication* and *division*.

The logical operators *not* and *or* are supported as well. These operators together are functionally complete [7]. This means that every truth table can be expressed using an term consisting of only these two operators. Other logical operators like *and* and *xor* are therefore redundant and can be omitted.

Equality and inequality can be checked using the operators $=$ and $<$. Most of the other, normally supported operators in other programming languages are not supported. For instance the greater than operation ($>$), the less and equal operation ($\leq$) are not supported. These operators can be defined using or desugared into the already present operators.

Increasing the expressive power of the language are the function application and the if-then-else expression.

Refinements that can be applied to the types are kept relatively simple. The type checker has to solve constraints based on these refinements. In order to keep these constraints decidable and solvable in reasonable time, the predicates are limited to boolean expressions without function applications and if-then-else expressions, that is the only difference in this language between predicates and

⟨*program*⟩ ::= ⟨*function*⟩*

⟨*function*⟩ ::= **fun** ⟨*id*⟩ (⟨*param*⟩*) → ⟨*type*⟩ = ⟨*expr*⟩

⟨*expr*⟩ ::= **not** ⟨*expr*⟩
| ⟨*expr*⟩ **or** ⟨*expr*⟩
| ⟨*expr*⟩ * ⟨*expr*⟩
| ⟨*expr*⟩ / ⟨*expr*⟩
| ⟨*expr*⟩ + ⟨*expr*⟩
| ⟨*expr*⟩ - ⟨*expr*⟩
| ⟨*expr*⟩ = ⟨*expr*⟩
| ⟨*expr*⟩ < ⟨*expr*⟩
| **if** ⟨*expr*⟩ **then** ⟨*expr*⟩ **else** ⟨*expr*⟩
| ⟨*id*⟩ (⟨*expr*⟩*)
| ⟨*id*⟩
| ⟨*const*⟩

⟨*type*⟩ ::= ⟨*base*⟩ ⟨*refinement*⟩

⟨*base*⟩ ::= **int** | **bool**

⟨*refinement*⟩ ::= { ⟨*pred*⟩ }

⟨*pred*⟩ ::= **not** ⟨*pred*⟩
| ⟨*pred*⟩ **or** ⟨*pred*⟩
| ⟨*pred*⟩ * ⟨*pred*⟩
| ⟨*pred*⟩ / ⟨*pred*⟩
| ⟨*pred*⟩ + ⟨*pred*⟩
| ⟨*pred*⟩ - ⟨*pred*⟩
| ⟨*pred*⟩ = ⟨*pred*⟩
| ⟨*pred*⟩ < ⟨*pred*⟩
| ⟨*id*⟩
| ⟨*const*⟩

⟨*const*⟩ ::= ⟨*int*⟩ | **true** | **false**

⟨*param*⟩ ::= ⟨*id*⟩ : ⟨*type*⟩

⟨*id*⟩ ::= [a-z] [a-z 0-9]*

⟨*int*⟩ ::= $[0\text{-}9]^+$

Figure 1: Syntax of Core Language

```
fun one() → int { v = 1 } = 1

fun div(a: int, b: int { v ≠ 0 })
    → int = a / b

fun min(a: int, b: int)
    → int { v = a or v = b } =
        if a < b then a else b

fun fib(n: int { v ≥ 0 })
    → int { v ≥ 0 } =
        if n < 2 then n
        else fib(n − 2) + fib(n − 1)
```

Figure 2: Sugared Code Examples

expressions. This leaves us with a set of possible first-order predicates that are quantifier-free. It is known that these predicates are decidable and therefore checkable at compile time [3].

To illustrate, some code samples in 2 are given that can be written in the language, considering that some desugaring is allowed.

III. Type System

In other papers that discuss refinement types, a rather theoretical approach is taken with regard to the type system. On a very fundamental level is reasoned about types, functions and expressions. In this paper the approach is flipped in a sense. The first step in the research was to have a working fully featured refinement-type-checker. The next step was to translate the algorithmic implementation to a more declarative implementation with typing rules. This way the typing rules are close to an actual implementation.

The pipeline of the system is: parsing → base type checking → refinement checking → potential interpretation. Interpretation is not implemented as it is out of scope of the paper.

The first step is lexing and parsing. For this a parser-generator was used, namely ANTLR [8]. Any other form of parsing is naturally correct as long the resulting abstract syntax tree conforms with the core language syntax in 1.

The second step is the base type checker. This step was done separately from the refinement type because both checkers effectively check different things. The base-checker actually checks the types and the refinement-checker checks that the refinements hold. Since the scope of this paper focuses more on the refinement-checker, it means that the

base typing rules are not discussed. The rules are worked out and included in 3 for referencing purposes. One thing to note in the base-type-checker is that the refinements, the predicates are disregarded. This shows an important notion of refinement types. Refinement types do not give more power to a language, a refinement-type-system is a conservative extension [1] to a conventional type-system. Any program with refinements (and that type-checks) works in a system without refinements as well, the reverse does not hold.

The third step is refinement checking. During this check it is assumed that all types hold in the program. So, all refinements actually evaluate to a boolean and all other expressions type check as well. The typing rules are shown in 4 and 5. Some rules are very repetitive. For example the ADD and SUB in 5 are almost identical. Only the more radical typing rules will be discussed the others are expected to become clear from explanation from similar typing rules.

i. Context Variants

Before discussing the typing rules it is convenient to discuss the different types of contexts that are used. In conventional type-systems and their rules the context is conventionally annotated with: Γ. The context contains function descriptions and the types of certain variables or parameters, not dissimilar in the typing rules of the base checker in this language. In the refinement checker, Γ is used differently. The context still contains, and only contains, function descriptions. Therefore, Γ contains function descriptions but not variables and parameters. Do note that the function descriptions contain the names of the parameters as well. This is needed later on to type check dependent types. Also note that not the types of the parameters and the return type are part of the function description only the refining predicates. The types have already been checked by the base-type-checker and the types are assumed to be correct.

The second context used is the context annotated with Θ. This context maps variables of the language to created variables in the constraint solver. A large portion of the refinement typing rules is the translation of the expressions and predicates to something a constraint solver can understand. The constraint solver will collect all the assertions while type-checking and determine if the refinements are actually valid. If that is not the case then checking fails and the program is invalid and rejected.

A constraint solver is needed is therefore needed, the implementation uses a constraint solver developed by Microsoft called Z3. Z3 implements the SMT-LIB version 2.0 standard [9]. This is a standard on how to declare, define and encode constraints. Multiple constraints solvers implement this standard and therefore it constraint solving does not rely on specific features only present in Z3. Other solvers that implement the same standard could be used as well, like AProVe [10], CVC4 [11] and Yices [12]. Coming back to the context Θ, this maps the variables from the language to the variables used by the constraint solver. These variable can then be substituted when translating the expressions and predicates.

The third and final context is annotated with Φ. This context contains assertions, assertions that can be interpreted by the constraint solver. The assertions are all boolean expressions or predicates and the constraint solver can be asked if the assertions and constraints can be satisfied. If it can be satisfied then a model is returned with possible evaluations of the variables such that the constraints hold.

ii. Typing Rules

With this elementary understanding of the different variants of contexts used the typing rules can be discussed. The root of the typing rules is the PROG rule in 4. The rule checks that a program is well-formed, indicated with the $\diamond$ symbol, which means that all the functions are well-formed. The first premise in the rules sequents collects the function descriptions as discussed earlier. Therefore, function type with the parameter's names as well. This context is immutable for the rest of the checking procedure, since only global functions are allowed and all are known at compile time. The other premises of the rule check that every function is well-formed.

The PROG rule checks that every function is well-formed. That means that given the contexts Γ, Θ and Φ containing the function descriptions, the parameters mapping to solver variables and the constraints that the body expression of the function must have the refinement given to the return value of the function. The FUN rule in 4 checks this property. There are no Θ and Φ contexts yet, these will be created with the parameters of the function, If there are none then they will naturally be empty. If there are parameters then they will translated to

Well Typed Program Rules:

$$\frac{\begin{array}{c}\Gamma = f_0 := (x_{00} : t_{00}\ \ldots\ x_{0n} : t_{0n}) \to t_0,\ \ldots,\ f_q := (x_{q0} : t_{q0}\ \ldots\ x_{qm} : t_{qm}) \to t_q \\ \Gamma \vdash \textbf{fun}\ f_0\ (x_{00} : t_{00}\ \ldots\ x_{0n} : t_{0n}) \to t_0 = e_0 : \diamond \\ \vdots \\ \Gamma \vdash \textbf{fun}\ f_q\ (x_{q0} : t_{q0}\ \ldots\ x_{qm} : t_{qm}) \to t_q = e_q : \diamond\end{array}}{\left\{\begin{array}{l}\textbf{fun}\ f_0\ (x_{00} : t_{00}.p_{00}\ \ldots\ x_{0n} : t_{0n}.p_{0n}) \to t_0.p_0 = e_0 \\ \vdots \\ \textbf{fun}\ f_q\ (x_{q0} : t_{q0}.p_{q0}\ \ldots\ x_{qm} : t_{qm}.p_{qm}) \to t_q.p_q = e_q\end{array}\right\} : \diamond}\ \textsc{BaseProg}$$

Well Typed Function Rules:

$$\frac{\Gamma,\ x_0 : t_0,\ \ldots,\ x_n : t_n \vdash e : t}{\Gamma \vdash \textbf{fun}\ f\ (x_0 : t_0\ \ldots\ x_n : t_n) \to t = e : \diamond}\ \textsc{BaseFun}$$

Well Typed Expression Rules:

$$\frac{\Gamma \vdash e_0 : \textbf{int} \quad \Gamma \vdash e_1 : \textbf{int}}{\Gamma \vdash e_0 * e_1 : \textbf{int}}\ \textsc{BaseMult} \qquad \frac{\Gamma \vdash e_0 : \textbf{int} \quad \Gamma \vdash e_1 : \textbf{int}}{\Gamma \vdash e_0 / e_1 : \textbf{int}}\ \textsc{BaseDiv}$$

$$\frac{\Gamma \vdash e_0 : \textbf{int} \quad \Gamma \vdash e_1 : \textbf{int}}{\Gamma \vdash e_0 + e_1 : \textbf{int}}\ \textsc{BaseAdd} \qquad \frac{\Gamma \vdash e_0 : \textbf{int} \quad \Gamma \vdash e_1 : \textbf{int}}{\Gamma \vdash e_0 - e_1 : \textbf{int}}\ \textsc{BaseSub}$$

$$\frac{\Gamma \vdash e_0 : \textbf{int} \quad \Gamma \vdash e_1 : \textbf{int}}{\Gamma \vdash e_0 < e_1 : \textbf{bool}}\ \textsc{BaseLess} \qquad \frac{\Gamma \vdash e_0 : t \quad \Gamma \vdash e_1 : t}{\Gamma \vdash e_0 = e_1 : \textbf{bool}}\ \textsc{BaseEq}$$

$$\frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash \textbf{not}\ e : \textbf{bool}}\ \textsc{BaseNot} \qquad \frac{\Gamma \vdash e_0 : \textbf{bool} \quad \Gamma \vdash e_1 : \textbf{bool}}{\Gamma \vdash e_0\ \textbf{or}\ e_1 : \textbf{bool}}\ \textsc{BaseOr}$$

Well Typed Constant Rules:

$$\frac{}{\Gamma \vdash n : \textbf{int}}\ \textsc{BaseNum} \qquad \frac{}{\Gamma \vdash \textbf{true} : \textbf{bool}}\ \textsc{BaseTrue} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool}}\ \textsc{BaseFalse}$$

Well Typed Variable Rules:

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textsc{BaseVar}$$

Well Typed Special Expression Rules:

$$\frac{\Gamma(f) = (t_0\ \ldots\ t_0) \to t \quad \Gamma \vdash e_0 : t_0 \quad \cdots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash f\ e_0\ \ldots\ e_n : t}\ \textsc{BaseApp}$$

$$\frac{\Gamma \vdash e_c : \textbf{bool} \quad \Gamma \vdash e_0 : t \quad \Gamma \vdash e_1 : t}{\Gamma \vdash \textbf{if}\ e_c\ \textbf{then}\ e_0\ \textbf{else}\ e_1 : t}\ \textsc{BaseITE}$$

Figure 3: Base Typing Rules

Well Formed Program Rules:

$$\frac{\begin{array}{c}\Gamma = f_0 := (x_{00} : p_{00}\ \ldots\ x_{0n} : p_{0n}) \to p_0,\ \ldots,\ f_q := (x_{q0} : p_{q0}\ \ldots\ x_{qm} : p_{qm}) \to p_q \\ \Gamma \vdash \textbf{fun}\ f_0\ (x_{00} : p_{00}\ \ldots\ x_{0n} : p_{0n}) \to p_0 = e_0 : \diamond \\ \vdots \\ \Gamma \vdash \textbf{fun}\ f_q\ (x_{q0} : p_{q0}\ \ldots\ x_{qm} : p_{qm}) \to p_q = e_q : \diamond\end{array}}{\left\{\begin{array}{l}\textbf{fun}\ f_0\ (x_{00} : t_{00}.p_{00}\ \ldots\ x_{0n} : t_{0n}.p_{0n}) \to t_0.p_0 = e_0 \\ \vdots \\ \textbf{fun}\ f_q\ (x_{q0} : t_{q0}.p_{q0}\ \ldots\ x_{qm} : t_{qm}.p_{qm}) \to t_q.p_q = e_q\end{array}\right\} : \diamond}\textsc{Prog}$$

Well Formed Function Rules:

$$\frac{x_0 : p_0\ \ldots\ x_n : p_n \Rightarrow \Theta; \Phi \quad \Gamma; \Theta; \Phi \vdash e : p}{\Gamma \vdash \textbf{fun}\ f\ (x_0 : p_0\ \ldots\ x_n : p_n) \to p = e : \diamond}\textsc{Fun}$$

$$\frac{\begin{array}{c}\cdot \vdash x_0 : p_0 \Rightarrow x_0'\ ;\ \phi_0 \\ x_0 := x_0' \vdash x_1 : p_1 \Rightarrow x_1'\ ;\ \phi_1 \\ \vdots \\ x_0 := x_0',\ \ldots,\ x_{n-1} := x_{n-1}' \vdash x_n : p_n \Rightarrow x_n'\ ;\ \phi_n\end{array}}{x_0 : p_0\ \ldots\ x_n : p_n \Rightarrow x_0 := x_0',\ \ldots,\ x_n := x_n'\ ;\ \phi_0,\ \ldots,\ \phi_n}\textsc{Params}$$

$$\frac{\Theta,\ x := x' \vdash p \rightsquigarrow \phi}{\Theta \vdash x : p \Rightarrow x'; \phi}\textsc{Param} \qquad \frac{\Gamma; \Theta; \Phi \vdash e : p \Rightarrow \nu; \Phi'}{\Gamma; \Theta; \Phi \vdash e : p}\textsc{Base}$$

Well Formed Expression Rules:

$$\frac{\Theta \vdash \nu : p \Rightarrow \phi \quad \Gamma; \Theta; \Phi \vdash e \Rightarrow e'; \Phi' \quad \Phi,\ \Phi',\ \nu = e' \vdash \neg\exists m.m \vDash \neg\phi}{\Gamma; \Theta; \Phi \vdash e : p \Rightarrow \nu;\ \Phi', \nu = e'}\textsc{Hold}$$

$$\frac{\Theta,\ v := \nu \vdash p \rightsquigarrow \phi}{\Theta \vdash \nu : p \Rightarrow \phi}\textsc{Ref}$$

Figure 4: Refinement Typing Rules Part 1

contexts with the PARAMS rule in 4.

The PARAMS rule starts with an empty Θ context for the first parameter and translates the parameter with the PARAM rule, in 4, to a solver variable and an assertion. Please note that the refinement-checker assumes that there is no such thing as a base type, meaning that every type is a refinement type. The base type can be given by giving a tautology as predicate. For instance *int* $\{v = v\}$ is equivalent to the base type *int*. The translated solver variable is collected and used as given Θ context for the second parameter. This allows for parameters that are dependent on earlier parameters. A function description like: **fun** zot(a: **int** $\{v < 3\}$, b: **int** $\{v <$ a$\}$) is therefore possible. This process continues until the last parameter. All the generated solver variables are collected into a Θ context and all the generated constraints are collected into a Φ context. These contexts are the ones that are returned by the PARAMS rule.

ii.1 Hold Typing Rule

With the initial contexts in place the BASE rule in 4 can be applied. This is actually a HOLD rule, in 4, with the returned elements omitted. The HOLD rule is discussed instead. What the HOLD rule does is checking if the expression e has the refinement p given the contexts Γ, Θ and Φ. The rule returns the generated solver variable ν and a Φ context with assertions learned when traversing the abstract syntax tree of e. The rule has three premises.

The first premise uses the REF rule to generate an assertion ϕ while given a Θ context and a unique solver variable ν. In the algorithmic implementation a solver variable is created with a unique name by using a counter and appending that number to the string "_". So, example variable names are "_0", "_1" and "_123". Naturally there are other correct strategies. It is only imperative that the variables are unique. The REF rule generates the constraint by temporarily adding $v = \nu$ to the Θ context and then substituting the variables in the predicate with their matching solver variables. Note the difference between v and ν. v is the variable written in the language that indicates the value of the type and ν (nu) is the solver variable that is uniquely generated to represent the value of that type in the constraint solver. Substitution is indicated with the $\rightsquigarrow$ symbol. So, $\Theta \vdash p \rightsquigarrow \phi$ states that the variables in p are substituted with the matching variable in Θ into the assertion ϕ. This simple substitution is possible as a consequence of the choice to keep the predicates simple. Recall that this was done to keep the predicates decidable. The predicates that can be expressed in the core language can be expressed too in the constraint solver. Therefore only substitution of variables is needed.

The second premise of the HOLD rule translates the expression e into an alternate expression e' understandable by the constraint solver. Additional assertions are returned as well. The responsible typing rules are discussed later in the section.

The third premise actually checks that the refinement of the return type holds with the constraint solver. The premise looks like this: Φ, Φ', $\nu = e' \vdash \neg\exists m.m \vDash \neg\phi$. Given the Φ context given in the conclusion of the HOLD rule, the returned assertions of the expression translation (Φ') and the additional constraint that the return value ν is equal to the translated expression e', the solver is asked if there is not a model m such that that model m satisfies ($\vDash$) the inverse of the generated constraint ϕ by the REF rule. A model here is an evaluation of the generated solver variables that makes all the given assertions hold. If a model, that makes the inverse of ϕ hold, does not exist then the function is well-formed, if it does exist then the function is not well-formed.

Some reasoning behind why this is the case: A constraint solver is given some assertions and can be asked to give a model such that the assertions hold. Proving a constraint is not supported by the SMT-LIB version 2.0 standard [9] and therefore not by Z3. To prove that given some assertions that another assertion is valid, meaning that it always holds if the given assertions hold, then the existence of a model that makes the inverse of the to be validated assertion hold should be unsatisfiable. If something is always true then it logically follows that the something is never false. Because the solver can not be asked whether something is always true it was chosen to ask if the inverse never holds. This can be asked of a constraint solver.

ii.2 Translation Rules

It was earlier conveniently assumed that some typing rules existed that translated an expression e to an alternative constraint-solver understandable expression e'. These typing rules are worked out and given in 5. To start with a simple example rule, take the typing rule MULT, this rule translates a multiplication expression. The premises trans-

Base Expression Translation Rules:

$$\frac{\Gamma;\Theta;\Phi \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\Phi \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash e_0 * e_1 \Rightarrow e_0' * e_1';\ \Phi_0,\Phi_1}\textsc{Mult} \quad \frac{\Gamma;\Theta;\Phi \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\Phi \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash e_0 / e_1 \Rightarrow e_0' / e_1';\ \Phi_0,\Phi_1}\textsc{Div}$$

$$\frac{\Gamma;\Theta;\Phi \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\Phi \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash e_0 + e_1 \Rightarrow e_0' + e_1';\ \Phi_0,\Phi_1}\textsc{Add} \quad \frac{\Gamma;\Theta;\Phi \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\Phi \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash e_0 - e_1 \Rightarrow e_0' - e_1';\ \Phi_0,\Phi_1}\textsc{Sub}$$

$$\frac{\Gamma;\Theta;\Phi \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\Phi \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash e_0 < e_1 \Rightarrow e_0' < e_1';\ \Phi_0,\Phi_1}\textsc{Less} \quad \frac{\Gamma;\Theta;\Phi \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\Phi \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash e_0 = e_1 \Rightarrow e_0' = e_1';\ \Phi_0,\Phi_1}\textsc{Eq}$$

$$\frac{\Gamma;\Theta;\Phi \vdash e \Rightarrow e';\Phi'}{\Gamma;\Theta;\Phi \vdash \textbf{not}\ e \Rightarrow \textbf{not}\ e';\ \Phi'}\textsc{Not} \quad \frac{\Gamma;\Theta;\Phi \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\Phi \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash e_0\ \textbf{or}\ e_1 \Rightarrow e_0'\ \textbf{or}\ e_1';\ \Phi_0,\Phi_1}\textsc{Or}$$

Constant and Variable Translation Rules:

$$\frac{\Theta(x) = x'}{\Gamma;\Theta;\Phi \vdash x \Rightarrow x';\cdot}\textsc{Var} \quad \frac{}{\Gamma;\Theta;\Phi \vdash n \Rightarrow n;\cdot}\textsc{Num}$$

$$\frac{}{\Gamma;\Theta;\Phi \vdash \textbf{true} \Rightarrow \textbf{true};\cdot}\textsc{True} \quad \frac{}{\Gamma;\Theta;\Phi \vdash \textbf{false} \Rightarrow \textbf{false};\cdot}\textsc{False}$$

Special Expression Translation Rules:

$$\frac{\begin{array}{c}\Gamma(f) = (x_0 : p_0\ \ldots\ x_n : p_n) \to p \\ \Gamma;\Theta;\Phi \vdash e_0 : p_0 \Rightarrow x_0';\Phi_0 \\ \vdots \\ \Gamma;\Theta;\Phi \vdash e_n : p_n \Rightarrow x_n';\Phi_n \\ \Theta,\ x_0 := x_0',\ \ldots,\ x_n := x_n' \vdash x' : p \Rightarrow \phi\end{array}}{\Gamma;\Theta;\Phi \vdash f\ e_0\ \ldots\ e_n \Rightarrow x';\ \Phi_0,\ldots,\Phi_n,\phi}\textsc{App}$$

$$\frac{\Gamma;\Theta;\Phi \vdash e_c \Rightarrow e_c';\Phi_c \quad \Gamma;\Theta;\ \Phi,\Phi_c,e_c' \vdash e_0 \Rightarrow e_0';\Phi_0 \quad \Gamma;\Theta;\ \Phi,\Phi_c,\neg e_c' \vdash e_1 \Rightarrow e_1';\Phi_1}{\Gamma;\Theta;\Phi \vdash \textbf{if}\ e_c\ \textbf{then}\ e_0\ \textbf{else}\ e_1 \Rightarrow \textbf{if}\ e_c'\ \textbf{then}\ e_0'\ \textbf{else}\ e_1';\ \Phi_c,\Phi_0,\Phi_1}\text{ITE}$$

Figure 5: Refinement Typing Rules Part 2

late both the sub-expressions and return both new assertions. So, if the expression is: $e_0 * e_1$ then the premises translate e_0 to e'_0 and an assertion context Φ_0 and e_1 to e'_1 and an assertion context Φ_1. The conclusion then returns the multiplication of the e'_0 and e'_1 and the combination of Φ_0 and Φ_1. Thus, the returned expression is $e'_0 * e'_1$ and the returned assertion context is Φ_0, Φ_1. A large collection of the other rules work in a similar way. For instance but not limited by Div, Less, Not. These very similar typing rules are under the header "Base Expression Translation Rules" in 5 and are not discussed further.

There are three typing rules that are similar to each other as well. The typing rules Num, True, False under the header "Constant and Variable Translation Rules" in 5. They are similar because they do not have any sequents. The expression **true** always translates to **true** with an empty additional Φ context $(\cdot)$. Something not dissimilar holds for the other two rules.

A variable x is translated to a solver variable x' with an empty Φ context if the variable x is matched to x' in the Θ context. The Var typing rule is used for this.

Now only the more interesting function application expression and if-then-else expression translation rules remain to be discussed. Lets look at the function application translation first, the App typing rule. Function application is supported in a sense in constraint solvers however function applications of recursive functions are not. To exclude recursive functions would make the language less powerful. It was therefore decided to develop a translating procedure that would allow for recursive functions and their applications. In a nutshell, the arguments are checked to hold the refinement put upon the parameters of the called function and the entire function application is then substituted with a solver variable that has the same type as the return type of the applied function.

The first premise in the App rule is to retrieve the function description of *f*. Since the base checker already succeeded, this step always succeeds too.

The next steps is to check every argument of the function application if the refinement of its fellow parameter is obeyed. This is checked by using the earlier discussed Hold typing rule. The already given contexts Γ, Θ and Φ are also given to these rules and if correct, a solver variable and additional assertions are returned by each rule.

The third premise collects the parameters and the matching solver variables combines these with the given Θ context. This new context is used to translate the returned refinement of the function to an assertion. Note that there is a circular relation between the typing rules Hold and the translating rules. The Hold rule translates the expression however the translating rule can use the Hold rule again in the App rule.

The last translating rule is that of the if-then-else expression. Not particularly difficult, though two concept need to be noted. The if-then-else expression is supported by the SMT-LIB version 2.0 standard [9]. Therefore it can be translated somewhat simple in that sense.

The condition expression is the first to be translated and results in a expression e'_c and a context Φ_c. Then the expression e_0 is translated. Note that the context Φ_c and the assertion e'_c is given to the given assertion context of the rule. Because the base type checker has already passed it can be assumed that the condition is a boolean expression which means that if it is translated that it can be used as an assertion in a Φ context. This knowledge is passed such that if an if expression check is done that in the then expression that knowledge is also known. For instance: **fun** baz(x: **int**)$\rightarrow$ **int** = **if** x $\neq$ 0 **then** div(3, x) **else** -1 , in the second parameter of *div*, is not allowed to be zero but there is no refinement on the parameter x so every value is possible including zero. The condition of the if-then-else expression however checks that x is not equal to zero. So in the then expression this can be assumed to hold as well. For this reason the condition assertion is added to the given Φ context when translating the then expression. The exact opposite is true in the else expression and therefore the inverse of the translated condition $(\neg e'_c)$ is added to that Φ context when translating e_1.

IV. Worked Out Example

It is difficult to understand the typing rules by explaining them in such an abstract manner. For this reason in this section an example will be worked out. The rule derivations are given and discussed in detail. The example, shown in 6, chosen is kept simple yet contains three important features. The first feature is dependent typing. The return type of the function *foo* is dependent on its parameter *a*. The second feature is a function application. The third feature is the learning process from if expressions. If the parameter b is smaller than 7 then $b + 1$ can not be smaller than 5, the refinement

```
fun foo(a: int { v < 5 })
    → int { v = a + 2 } = a + 2

fun bar(b: int { v < 7 })
    → int { v < 9 } =
        if b < 3 then foo(b + 1) + 3
        else 0
```

Figure 6: In Depth Example Code

upon the parameter of *foo*. However, because it was checked that b is less than 3 as well. Then $b + 1$ is smaller than 5.

i. Program Level

The program is first parsed and the base checker is then run with the resulting abstract syntax tree. The process of this is omitted as it is out of the scope of the paper.

After the base-type-checker has succeeded then the refinement checker is run. First the PROG typing rule is applied, see derivation (0) in 7, which first creates the Γ context out of the function descriptions. Recall that not the types but the refinements are used for the function description and that the parameter names are stored too. The resulting Γ context looks like this:

$$\Gamma = foo := (a : \{v < 5\}) \rightarrow \{v = a + 2\},$$
$$bar := (b : \{v < 7\}) \rightarrow \{v < 9\}$$

All other contexts are completely written out in the rule derivations, the Γ context is the exception. The context is immutable and standard at all the places where a Γ is needed. So, to save precious space only Γ is written. The evaluation of Γ is as shown above.

The two other sequents of (0) are the rule derivations (1) and (2). (1) checks if *foo* is well-formed and (2) checks if *bar* is well-formed.

ii. Checking Foo

Lets start with the derivation of *foo*. Derivation (1) is the application of the FUN rule. The first premise of the FUN rule creates a Θ and Φ context from the parameters of the function with the PARAMS rule. Since the function has only one parameter it means that the PARAM is only applied once with an empty given Θ context. In this case $a : v < 5$ is translated to the solver variable a' and the constraint $a' < 5$. All the generated solver variables and assertions are collected, only one of each in this case, and returned as a Θ context and a Φ context. The Θ context contains the matching $a := a'$ and the Φ context contains the assertion $a' < 5$. With the already known Γ context and the generated Θ and Φ contexts the BASE rule can be applied. Already mentioned is that the BASE rule is identical to the HOLD rule albeit the returned solver variable and Φ context are omitted.

So, rule derivation (3) in 7 is the HOLD rule applied for *foo*. The first premise applies the REF rule and uses a generated solver variable, in this case ν_0, to create the constraint which always needs to hold. In the premise of that REF rule v is matched to ν_0 in the Θ context and all the variables in the refinement are substituted with the matching solver variables. The means that $v = a + 2$ is translated to $\nu_0 = a' + 2$.

The second premise in derivation (3) is translating the expression. For this particular function it is not very interesting to discuss. All that is needed to know is that the expression $a + 2$ is translated to $a' + 2$ and that no additional assertions are returned $(\cdot)$. The translating rules are discussed in depth for the derivation of the *bar* function later on. The derivations of the translation are shown in derivation (4) in 7.

The third and last premise in (3) checks that given the assertions in Φ ($a' < 5$) and that the returned value ν_0 is equal to $a' + 2$ that there does not exist a model such that the model satisfies $\neg(\nu_0 = a' + 2)$. This is fairly straightforward that this is the case. It is impossible for ν_0 to be equal to $a' + 2$ and not to be equal to $a' + 2$. The constraint solver concludes as a consequence that the function *foo* is well-formed.

iii. Checking Bar

Now that the function *foo* is known to be well-formed only *bar* has to be checked for well-formedness. The rule derivation (2) in 8 checks this property, it is again an application of the FUN typing rule.

The derivations first premise is almost identical to that of rule derivation (1) in 7. The function has only one parameter, the parameters name is now b instead of a and the refinement is now $v < 7$ instead of $v < 5$. Otherwise, the derivation is practically identical. The derivation is therefore not discussed further.

The second premise of the derivation is again the application of the BASE rule which in turn applies

$$\Gamma = foo := (a : \{v < 5\}) \to \{v = a + 2\},\ bar := (b : \{v < 7\}) \to \{v < 9\}$$

(1) (2)

$$\frac{}{\left\{\begin{array}{l}\textbf{fun}\ foo\ (a : \textbf{int}\{v < 5\}) \to \textbf{int}\{v = a + 2\} = a + 2 \\ \textbf{fun}\ bar\ (b : \textbf{int}\{v < 7\}) \to \textbf{int}\{v < 9\} = \textbf{if}\ b < 3\ \textbf{then}\ foo(b + 1) + 3\ \textbf{else}\ 0\end{array}\right\} : \diamond}(0)$$

$$\frac{\dfrac{\dfrac{v := a' \vdash v < 5 \rightsquigarrow a' < 5}{\cdot \vdash a : \{v < 5\} \Rightarrow a'; a' < 5}}{a : \{v < 5\} \Rightarrow a := a'; a' < 5} \quad \dfrac{(3)}{\Gamma; a := a'; a' < 5 \vdash a + 2 : \{v = a + 2\}}}{\Gamma \vdash \textbf{fun}\ foo\ (a : \{v < 5\}) \to \{v = a + 2\} = a + 2 : \diamond}(1)$$

$$\frac{\dfrac{a := a', v := \nu_0 \vdash v = a + 2 \rightsquigarrow \nu_0 = a' + 2}{a := a' \vdash \nu_0 : \{v = a + 2\} \Rightarrow \nu_0 = a' + 2} \quad (4) \quad \dfrac{}{a' < 5, \nu_0 = a' + 2 \vdash \neg\exists m.m \vDash \neg(\nu_0 = a' + 2)}}{\Gamma; a := a'; a' < 5 \vdash a + 2 : \{v = a + 2\} \Rightarrow \nu_0; \nu_0 = a' + 2}(3)$$

$$\frac{\dfrac{\Theta(a) = a'}{\Gamma; a := a'; a' < 5 \vdash a \Rightarrow a'; \cdot} \quad \dfrac{}{\Gamma; a := a'; a' < 5 \vdash 5 \Rightarrow 5; \cdot}}{\Gamma; a := a'; a' < 5 \vdash a + 2 \Rightarrow a' + 2; \cdot}(4)$$

Figure 7: Workout of Example Part 1

the HOLD rule. The derivation of this HOLD rule is given in (5) in 8.

The first premise derivation is again an application of the REF rule. It is very similar to the similar derivation for the *foo* function. It is therefore not discussed in depth, only note that the generated solver variable is now ν_1 and that the generated constraint is that $\nu_1 < 9$.

The second premise is the translation of the expression of the function into a constraint solver understandable expression. The derivation is shown in (6) in 8.

Derivation (6) is an application of the ITE rule, which has three sequents. The first sequent translates the condition to an assertion. To avoid repetition this translation is not discussed as well. It is relatively straightforward that if b is matched to b' that the expression $b < 3$ is then translated to $b' < 3$. The derivation is given in derivation (7) in 8.

The second premise of (6) is derivation (8), also shown in 8. This derivation translate the expression $foo(b + 1) + 3$ to the expression $\nu_2 + 3$. Additionally the assertions: $\nu_3 = b' + 1, \nu_2 = \nu_3 + 2$ are returned. The function application has been substituted with a solver variable ν_3 and the assertions $\nu_3 = b' + 1, \nu_2 = \nu_3 + 2$ tell something about the value of this variable.

This translation is done by derivation (8). The second premise of the derivation trivially translate the expression 3 to 3 again. The first premise however translates the function application $foo(b + 1)$ to ν_2, derivation (10).

Derivation (10), a application of the APP rule, checks first the function matched with the function identifier. In this case the identifier *foo* is matched with $(a : v < 5) \to v = a + 2$. All but the last premise checks if the arguments holds their respective refinements, in this case it is only one argument. The derivation of the this check is given in (11).

Derivation (11) is again an application of the HOLD rule. It is not discussed in depth. Do note that the HOLD rule returns a solver variable representing the argument and an additional context with assertions. This is discarded by the BASE rule for the body of a function. It is however important for function arguments. The returned solver variable ν_3 and the assertions $\nu_3 = b' + 1$ are used for dependent typing. If the returned value of *foo* is dependent on a parameter, which it is, then that parameter is later substituted with the solver variable ν_3. The assertions are conveyed as well to assist the constraint solver. In addition note that the assertion $b' < 3$ was added to the Φ context when translating the parent if-expression. This assertion allows the required refinement $v < 5$ to hold.

The last premise in (10), derivation (12), translate the refinement of the return type of the function *foo* into an assertion. The REF rule is therefore applied again. Note that $a := \nu_2$ and $v := \nu_3$ are added to the given Θ context which allows for $v = a + 2$ to be translated into $\nu_2 = \nu_3 + 2$.

This completes the derivation of (10) which in turn completes the derivation of (8) which brings us to the last premise of derivation (6). This premise simply asserts that the expression 0 trivially translates to 0. Now, derivation (6) is complete as well.

Now all the information is in place to check whether the refinement on the return type of *bar* holds. From the parameters it is known that $b' < 3$, from the application of *foo* we know that $\nu_3 = b' + 1$ and that $\nu_2 = \nu_3 + 2$, from the translation of the body expression we know that $\nu_1 = \textbf{if } b' < 3 \textbf{ then } \nu_2 + 3 \textbf{ else } 0$ and from the first sequent in (5) we know that the constraint $\nu_1 < 9$ must hold. The same as the earlier mentioned strategy is used. It is checked whether there exists a model such that it satisfies all the given assertions and that satisfies the inverse of $\nu_1 < 9$. If that does not exist then, which is the case, then the function *bar* is well-formed.

V. Evaluation of Type Checker

Evaluation of the type-system and checker was done with a test suite [5]. The test suite contained real world examples like the *division* function, *min* function and the *fibonacci* function. The suite contained as well contrived examples like the examples in 6.

All the different features have been tested in depth for instance dependent typing. Where parameters are dependent on earlier parameters, return types are dependent on parameters and the application of such functions. Naturally, combinations are tested as well.

Other features include the if-then-else expressions and their learning capabilities and recursive functions like *fibonacci*.

In terms of time consumption, the largest test run was a program of around 20 simple functions which took around 200 milliseconds to complete. This was including parsing and the base checker.

VI. Related Work

A number of different research papers on refinement typing, refinement type systems and refinement checkers have been published. Yitzhak Mandelbaum, David Walker and Robert Harper published a paper on the theory of refinement typing [1]. In this paper a ML-style refinement typing system is developed and the power of such a system is indicated with a collection of examples [1].

Another inspiration for this paper was the by Hongwei Xi and Frank Pfenning [13]. This paper discusses dependent typing in practical programming. Another ML-style type system was developed, only with support for dependent types. Thus, types dependent on value of other variables.

Noam Zeilberger published some research notes on the principles of refinement typing [4]. These principles are discussed by refining the simply typed lambda calculus. There is a heavy focus on sub-typing and parametric typing.

Besides heavily theoretical papers there are also papers published discussing definition of refinement type systems for practical programming languages. For instance Typescript [14], ML [15], Ruby [16] and Jolie [17]. These papers however only talk about such a system in a very theoretical and abstract fashion. Actual implementation of such a system is not or barely discussed.

Another prominent implementation of refinement types was the refinement type-system Liquid-Haskell [18], a refinement type-system for Haskell.

VII. Conclusions

To end with some concluding remarks on the developed refinement-type-system and refinement-type-checker, it is apparent that such a system is potentially useful in decreasing bugs in software. It was also discussed that code complexity could be decreased and performance increased. It could therefore be interesting future work to develop this type system further. For instance increasing the expressiveness of the language with strings with string manipulation features and lists with accompanying features like appending, folding and mapping.

Also optimization of the practical implementation of the system could be future work. When a program is altered it does not necessarily mean that all functions have to be type checked again. If they are not altered then they could be cached. Perhaps other optimizations could improve performance of this system or similar systems in general.

$$\dfrac{\dfrac{\dfrac{v := b' \vdash v < 7 \rightsquigarrow b' < 7}{\cdot \vdash b : \{v < 7\} \Rightarrow b' ; b' < 7}}{b : \{v < 7\} \Rightarrow b := b' ; b' < 7} \quad \dfrac{(5)}{\Gamma ; b := b' ; b' < 7 \vdash \textbf{if } b < 3 \textbf{ then } foo(b+1) + 3 \textbf{ else } 0 : \{v < 9\}}}{\Gamma \vdash \textbf{fun } bar\ (b : \{v < 7\}) \to \{v < 9\} = \textbf{if } b < 3 \textbf{ then } foo(b+1) + 3 \textbf{ else } 0 : \diamond}(2)$$

$$\dfrac{\dfrac{b := b', v := \nu_1 \vdash v < 9 \rightsquigarrow \nu_1 < 9}{b := b' \vdash \nu_1 : \{v < 9\} \Rightarrow \nu_1 < 9} \quad (6) \quad \dfrac{\begin{array}{l} b' < 7, \nu_3 = b' + 1, \nu_2 = \nu_3 + 2, \\ \nu_1 = \textbf{if } b' < 3 \textbf{ then } \nu_2 + 3 \textbf{ else } 0 \end{array} \vdash \neg \exists m.m \vDash \neg(\nu_1 < 9)}{\begin{array}{r} \textbf{if } b < 3 \\ \Gamma ; b := b' ; b' < 7 \vdash \textbf{then } foo(b+1) + 3 : \{v < 9\} \Rightarrow \nu_1 ; \\ \textbf{else } 0 \end{array} \begin{array}{l} \nu_3 = b' + 1, \nu_2 = \nu_3 + 2, \\ \nu_1 = \textbf{if } b' < 3 \textbf{ then } \nu_2 + 3 \textbf{ else } 0 \end{array}}(5)$$

$$\dfrac{\begin{array}{l} (7) \\ \textbf{if } b < 3 \\ \Gamma ; \Theta ; \Phi \vdash \textbf{then } foo(b+1) + 3 \Rightarrow \\ \textbf{else } 0 \end{array} \quad \begin{array}{l} (8) \quad (9) \\ \textbf{if } b' < 3 \\ \textbf{then } \nu_2 + 3 ;\ \nu_3 = b' + 1, \nu_2 = \nu_3 + 2 \\ \textbf{else } 0 \end{array}}{}(6)$$

$$\dfrac{\dfrac{\Theta(b) = b'}{\Gamma ; b := b' ; b' < 7 \vdash b \Rightarrow b' ; \cdot} \quad \dfrac{}{\Gamma ; b := b' ; b' < 7 \vdash 3 \Rightarrow 3 ; \cdot}}{\Gamma ; b := b' ; b' < 7 \vdash b < 3 \Rightarrow b' < 3 ; \cdot}(7)$$

$$\dfrac{(10) \quad \dfrac{}{\Gamma ; b := b' ; b' < 7, b' < 3 \vdash 3 \Rightarrow 3 ; \cdot}}{\Gamma ; b := b' ; b' < 7, b' < 3 \vdash foo(b+1) + 3 \Rightarrow \nu_2 + 3 ; \nu_3 = b' + 1, \nu_2 = \nu_3 + 2}(8)$$

$$\dfrac{\Gamma(foo) = (a : \{v < 5\}) \to \{v = a + 2\} \quad (11) \quad (12)}{\Gamma ; b := b' ; b' < 7, b' < 3 \vdash foo(b+1) \Rightarrow \nu_2 ; \nu_3 = b' + 1, \nu_2 = \nu_3 + 2}(10)$$

$$\dfrac{\dfrac{b := b', v := \nu_3 \vdash v < 5 \rightsquigarrow \nu_3 < 5}{b := b' \vdash \nu_3 : \{v < 5\} \Rightarrow \nu_3 < 5} \quad (13) \quad \dfrac{}{b' < 7, b' < 3, \nu_3 = b' + 1 \vdash \neg \exists m.m \vDash \neg(\nu_3 < 5)}}{\Gamma ; b := b' ; b' < 7, b' < 3 \vdash b + 1 : \{v < 5\} \Rightarrow \nu_3 ; \nu_3 = b' + 1}(11)$$

$$\dfrac{\dfrac{\Theta(b) = b'}{\Gamma ; b := b' ; b' < 7, b' < 3 \vdash b \Rightarrow b' ; \cdot} \quad \dfrac{}{\Gamma ; b := b' ; b' < 7, b' < 3 \vdash 1 \Rightarrow 1 ; \cdot}}{\Gamma ; b := b' ; b' < 7, b' < 3 \vdash b + 1 \Rightarrow b' + 1 ; \cdot}(13)$$

$$\dfrac{b := b', a := \nu_3, v := \nu_2 \vdash v = a + 2 \rightsquigarrow \nu_2 = \nu_3 + 2}{b := b', a := \nu_3 \vdash \nu_2 : \{v = a + 2\} \Rightarrow \nu_2 = \nu_3 + 2}(12)$$

$$\dfrac{}{\Gamma ; b := b' ; b' < 7, \neg(b' < 3) \vdash 0 \Rightarrow 0 ; \cdot}(9)$$

Figure 8: Workout of Example Part 2

References

[1] Y. Mandelbaum, D. Walker, and R. Harper, "An effective theory of type refinements," *SIGPLAN Not.*, vol. 38, pp. 213–225, Aug. 2003.

[2] H. Zhu, A. V. Nori, and S. Jagannathan, "Learning refinement types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, (New York, NY, USA), pp. 400–411, ACM, 2015.

[3] N. Vazou, A. Bakst, and R. Jhala, "Bounded refinement types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, (New York, NY, USA), pp. 48–61, ACM, 2015.

[4] N. Zeilberger, "Principles of type refinement (oplss 2016)," 2016.

[5] F. C. Slothouber, "Sorbet." `https://github.com/wollemat/sorbet`, 2019.

[6] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for typescript," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 310–325, ACM, 2016.

[7] Y.-D. Lyuu, "Functional completeness," 2016.

[8] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[9] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard - version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories, Edinburgh, Scotland,(SMT '10)*, 2010.

[10] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann, "Analyzing program termination and complexity automatically with aprove," *Journal of Automated Reasoning*, vol. 58, pp. 3–31, Jan 2017.

[11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, July 2011. Snowbird, Utah.

[12] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)* (A. Biere and R. Bloem, eds.), vol. 8559 of *Lecture Notes in Computer Science*, pp. 737–744, Springer, July 2014.

[13] H. Xi and F. Pfenning, "Dependent types in practical programming," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, (New York, NY, USA), pp. 214–227, ACM, 1999.

[14] P. Vekris, B. Cosman, and R. Jhala, "Refinement types for typescript," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 310–325, ACM, 2016.

[15] T. Freeman and F. Pfenning, "Refinement types for ml," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), pp. 268–277, ACM, 1991.

[16] M. Kazerounian, N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak, "Refinement types for ruby," *CoRR*, vol. abs/1711.09281, 2017.

[17] A. Tchitchigin, L. Safina, M. Mazzara, M. Elwakil, F. Montesi, and V. Rivera, "Refinement types in jolie," *CoRR*, vol. abs/1602.06823, 2016.

[18] N. Vazou, E. L. Seidel, and R. Jhala, "Liquidhaskell: Experience with refinement types in the real world," in *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, (New York, NY, USA), pp. 39–51, ACM, 2014.