Victoria University of Wellington
School of Engineering and Computer Science

SWEN430: Compiler Engineering

Assignment 2 — Type Checking

Due: Monday 9th August @ 23:59

Overview

This assignment will extend the WHILE language compiler with *unique references*. Unique references have a long history in the research literature [1–6], and have even found their their way into several mainstream programming languages. For example, C++11 introduced `unique_ptr` as a *smart pointer* which replaced the aged `auto_ptr` [7]. More recently, interest in the Rust language has been growing and many researchers are exploring the benefits of its strong type system [8–14]. This assignment takes inspiration from Rust and involves (amongst other things) developing a simplified form of *borrow checker* for WHILE.

You can download the latest version of the compiler from the SWEN430 homepage, along with the supplementary code associated with this assignment. *We recommend that you start from a fresh version of the compiler, rather than your extension from assignment 1.*

NOTE!!! the language and compiler has been updated since Assignment 1 with reference types.

Unique References

WHILE contains the concept of *references* (roughly similar to pointers in C), as the following illustrates:

```
void main() {
  &int p = new 123; // Allocate memory
  assert *p == 123;
  delete p; // Deallocate memory
}
```

In this assignment you will extend WHILE with a new form of reference `&T:1` called a *unique reference*. The intention is that a variable of such type (e.g. `&int:1`) is the *only* reference to that variable (i.e. it is *unique*). The following (valid program) illustrates:

```
void main() {
  &int:1 p = new 123;
  assert *p == 123;
  delete p;
}
```

Here, `new 1` returns a value of type `&int:1` since it is freshly allocated. Furthermore, `delete` will only accept a unique reference as its argument (i.e. because anything else would be unsafe).

There are several important requirements regarding safe use of unique references. For example, the following program is invalid:

```
void main() {
  &int:1 p = new 1;
  &int:1 q = p;
  assert *p == *q;
}
```

The above cannot be permitted because, otherwise, both `p` and `q` would refer to the same heap location — thus violating the invariant of type `&int:1`. Likewise, the following program is invalid:

```
void main() {
  &int:1 p = new 123;
  delete p;
  assert *p == 123;
}
```

This program is invalid because it attempts to use a reference after it has been deallocated. Finally, the following program is also invalid:

```
void main() {
  &int p = new 123;
  &int q = p;
  delete p;
}
```

This program is invalid because `delete` expects a unique reference but, in this case, is given a general reference (i.e. one which is not necessarily the only reference to the heap location in question).

1 Part 1 — Syntactic Extensions (5%)

The first part of this assignment is to extend the parser and abstract syntax tree to support unique references. The grammar is given as follows:

```
UniqueReferenceType ::= ReferenceType : 1
```

Having completed this, programs such as the following should parse correctly:

```
void main() {
  &int:1 p = new 123;
  &int q = p;
  delete p;
}
```

Observe that this program is actually invalid. However, we don't expect the compiler to be able to determine this yet!

2 Part 2 — Uniqueness Typing (40%)

Checking unique references are used safely is done in two states: *typing* and *analysis*. In the typing stage, an extended type checker ensures (amongst other things) correct use of subtyping involving unique references. For example, the following fails type checking because `&int:1` does not subtype of `int`:

```
&int:1 p = new 123;
int q = p;
```

This part of the assignment is concerned only with implementing an extended type checker for unique references. In particular, the following rules should be implemented:

- **Memory Allocation / Deallocation**. A `new` expression always returns a unique reference, whilst `delete` always requires a unique reference.

- **Reference Subtyping**. A unique reference `&T:1` is a subtype of a general reference of type `&T`. Thus, the following is permitted:

```
void main() {
  &int:1 p = new 123;
  &int q = p;
}
```

  Observe, however, that the opposite is *not* true. This means, for example, we cannot directly assign a generate reference to a unique reference. Thus, the following program is invalid:

```
void f(&int p) { &int:1 q = p; }
```

- (**Down casting**). The ability to convert a general reference back into a unique reference is critical for usability. However, this is an *unsafe* operation as it relies on the programmer's knowledge that a general reference is, in fact, unique. To reinforce this, the programmer is required to write a cast in such situations. The following (valid program) illustrates:

```
&int:1 create() {
  &int p = new 123;
  return (&int:1) p;
}
```

  This program is safe because the programmer knows that `p` is, in fact, unique (i.e. despite the fact that its type is `&int`).

In this part of the assignment, you should be focusing mainly on extending the `TypeChecker` class with the above rules. Keep in mind that some programs which use unique references incorrectly will type check. To catch such programs, we need a more advanced analysis (see later).

Relaxed subtyping. To ensure the type system of WHILE is *sound*, subtyping between general references was not permitted. For example, the following program is invalid:

```
void main() {
   &{int x, int y} p = new {x:123,y:223};
   &{int x} q = p;
   *q = {x:0};
   assert p->y == 223;
}
```

The above cannot be permitted as, otherwise, `p` would no longer reference a heap location of type `{int x, int y}` at the `assert` statement. Whilst the limitation on subtyping for WHILE is entirely sensible it can, in fact, be relaxed in the present of unique references. For example, the following should be valid:

```
void main() {
   &{int x, int y}:1 p = new {x:123,y:223};
   &{int x}:1 q = p;
   *q = {x:0};
}
```

Observe that, unlike before, we cannot attempt to access `p->y` after the final statement because `p` has been moved into `q`.

3 Part 3 — Uniqueness Analysis (55%)

The `TypeChecker` operates in a relatively simplistic fashion and, as a result, cannot catch all kinds of error. For example, the following program is invalid but still passes type checking:

```
 void main() {
  &int:1 p = new 123;
  &int:1 q = p;
  &int:1 r = p;
 }
```

The problem here is that `p` is used *twice* which violates the invariant that `q` is the only reference to the given heap location. Another example is the following:

```
void main() {
 &int:1 p = new 123;
 &int:1 q = p;
 assert *p == 123;
}
```

Again, the problem here is that `p` is used after being assigned to `q`. In the terminology of Rust, one would say that `p` is used after it has been *moved* to `q`. More specifically, after a unique reference has been *consumed* it should be considered *undefined*. A key challenge is what it means here to be consumed. Clearly, in the example above, when `p` is assigned to `q` it has been consumed. However, there are situations where `p` might be used on temporarily and, thus, should not be considered consumed. The following illustrates one such example:

```
void main() {
  &int:1 p = new 123;
  &int:1 q = new 223;
  assert p != q;
  assert *p == 123;
}
```

In the above, we see that `p` is used within the equality expression *without being consumed*. This makes sense as, once the equality expression is complete, `p` remains the only reference to the heap location in question. Another example is the following:

```
void main() {
  &{int f}:1 p = new {f:123};
  int i = p->f;
  assert p->f == 123;
}
```

Again, `p` is used within the expression `p->f` without being consumed. For this part of the assignment two classes have been provided to help you get started: `ConsumptionAnalysis` and `UniquenessChecker`.

Submission

Your assignment solution should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable*. See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code*.

3. **The testing mechanism supplied with the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. However, this does not prohibit you from adding new tests. *This is to ensure the automatic marking script can test your code*.

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information*.

Note: Failure to meet these requirements could result in your submission being reject by the submission system and/or zero marks being awarded.

Assessment

Marks for this assignment will be awarded based on the number of passing tests, as determined by the *automated marking system*. Furthermore, it is expected that all of the original tests supplied with the WHILE compiler (`WhileLangTests`) continue to pass, and *marks will be deducted each test which fails*.

NOTE: the test cases used for marking will differ from those in the supplementary code provided for this assignment. However, the intention of these hidden tests is that they are largely equivalent to those provided.

References

[1] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359, 1990.

[2] David Walker and Kevin Watkins. On regions and linear types. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 181–192, 2001.

[3] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[4] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 311–330. ACM, November 2002.

[5] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 176–200, 2003.

[6] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness and immutability. In *TOOLS Europe 2008*, 2008.

[7] Bjarne Stroustrup. Thriving in a crowded and changing world: C++ 2006-2020. *Proc. ACM Program. Lang*, 4(HOPL):70:1–70:168, 2020.

[8] J. Toman, S. Pernsteiner, and E. Torlak. Crust: A bounded verifier for rust. pages 75–80, 2015.

[9] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the Rust typechecker using clp (t). pages 482–493. IEEE Computer Society Press, 2015.

[10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 66:1–66:34. ACM Press, 2018.

[11] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, page Article 147. ACM Press, 2019.

[12] Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. Redleaf: Towards an operating system for safe and verified firmware. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, page 37–44. ACM Press, 2019.

[13] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based verification for Rust programs. In *Programming Languages and Systems*, pages 484–514. Springer-Verlag, 2020.

[14] D. J. Pearce. A lightweight formalism for reference lifetimes andborrowing in rust. *ACM Transactions on Programming Languages and Systems*, 43(1), 2021.