COMPILING FROM A TYPED DIALECT OF SCHEME TO WEBASSEMBLY

An Honors Thesis Presented

By

CHRISTOPHER BRIAN RYBICKI

Approved as to style and content by:

**** Arjun Guha 05/14/20 08:34 ****
Chair

**** Marius R Minea 05/14/20 11:04 ****
Committee Member

**** Philip Sebastian Thomas 05/15/20 14:13 ****
Honors Program Director

ABSTRACT

Compilers are an important kind of software engineering tool studied within the programming language community, which serve to transform written code from one language into another. One particularly significant compilation target is WebAssembly: a recently standardized assembly language that has been designed with portability, efficiency, and modularity in mind. WebAssembly has been integrated within browsers in order to offer developers the ability to run more computationally expensive processes and algorithms as an extension of JavaScript, allowing for the increased speed and efficiency of web applications. Our goal in this work is to show that we can design WebAssembly-targeted compilers for general languages in a way which, unlike most production compilers, guarantees type safety through all compiler passes. By achieving this goal, we hope to allow users to write type-safe programs which can be executed in a wide range of environments, while allowing compiler developers to debug, maintain, and extend their compilers more easily through the guarantees provided by type safety. To this end, we have successfully developed a prototype compiler from a typed variation of Scheme to WebAssembly which utilizes a robust type checker to validate intermediate code transformations for correctness. This required designing and implementing several compiler passes, such as closure conversion, lambda lifting, and code generation, in a fashion which is type-safe and adaptable to the limited data types and memory mechanisms available within the WebAssembly execution environment.

1. Introduction

Compilers are fundamental to modern software engineering. As an evolving field that agglomerates algorithmic techniques and design patterns from the rest of computer science, the study and implementation of compilers are important because of their fundamental purpose. A compiler, itself a piece of software, transforms the code in which software is written into the low-level bits and bytes that can be run on computer hardware. Depending on the design, compilers can target different hardware platforms, such as the Intel chips built into many Windows and macOS computers, or the custom processors used in embedded electronics. LLVM is one of the most common low-level intermediate languages for compilers to target, as it supports generating machine code for over a dozen different instruction sets.

This is beginning to change since the introduction of WebAssembly in 2017 [1]. WebAssembly is an open standard which defines a binary instruction format that enables code to run on a wide range of devices and environments, with a particular focus on web browsers. One of the primary uses of WebAssembly is to improve the speed of web applications (such as online shopping, banking, or email) that must often be built on JavaScript. However, since JavaScript was not designed primarily with speed in mind, in many use cases the language's design serves as a bottleneck for developers interested in building high-performance applications. WebAssembly aims to solve this by offering developers the ability to run more expensive computational processes and algorithms on web pages within a stripped-down environment that gets executed alongside JavaScript when the page is loaded [1].

So far, WebAssembly has only been widely available as a compilation target for C, C++, and Rust through a tool named Emscripten which compiles languages with LLVM-based backends to WebAssembly [2]. Our goal is to show that we can design WebAssembly-targeted compilers for more general languages in a way which, unlike most production compilers, guarantees type safety through all compiler passes. By achieving this goal, we hope to allow users to write type-safe programs which can be executed in a wide range of environments, while allowing compiler developers to debug, maintain, and extend their compilers more easily through the guarantees provided by type safety. To this end, we have developed a prototype compiler from a typed variation of Scheme to WebAssembly which utilizes a robust type checker to validate each intermediate AST transformation for correctness.

In section 2, we discuss the significance of this project in the context of the WebAssembly ecosystem and the evolution of type-safety in compilers. In section 3, we provide a review of related literature. In section 4, we formally describe the syntax and semantics of our compiler's source language as well as its intermediate representation, and discuss the compiler's architecture in detail. In section 5 we show annotated examples of several pieces of code compiled end-to-end, and provide justification of the type safety of the compiler. In section 6, we finally conclude by reviewing the goals and what we achieved, and discussing possible avenues for future work.

Our compiler is available to use and adapt as an open-source project at https://github.com/Chriscbr/scheme-to-wasm.

2. Significance

As the amount of software code produced by developers and generated by tools multiplies over time, ensuring the validity of such code becomes an increasingly challenging task due to the expanding complexity of software requirements and implementations. Since compilers are a key component of most software development toolchains used by engineers, defects in a compiler can lead at best to inefficient or faulty program behavior, and at worst to compiled programs being used as attack vectors by malicious actors. Thus, it is important to be able to be able to validate the correctness of code produced by compilers. A type system is one kind of tool which allows us to formally prove certain properties about a program's code and its respective behavior [3]. While type systems are limited in the kinds of properties they can prove in order to keep them tractable, they are one of the most fundamental and frequently used components of modern language design, and are the primary way in which we try to show our compiler has been built safely.

WebAssembly, as our choice of target language, is still largely unexplored due to its recency, and understanding the intricacies of compiling to such a platform will help us understand which areas the language is most lacking in in terms of features and safety. Growth in the area of WebAssembly development and research is likely to increase following the World Wide Web Consortium's (W3C) announcement on December 5, 2019 that the WebAssembly specification is now an official web standard [4]. Furthermore, several proposals have since been made to extend the language with features such as reference types, tail call optimization, multiple return values, and even garbage collection [5]. Since WebAssembly is designed to be used as a compiler target for browsers as well as other environments, understanding the level of type safety that can be embedded while compiling to WebAssembly is important for both type theorists and language implementors.

As we explain in section 3, when WebAssembly code is run, a validation step is performed to check if it satisfies several well-formedness conditions [6]. However, since WebAssembly has a simple type system, it does not guarantee (without sufficient correctness of the compiler transformations used to generate that code) that the generated WebAssembly code will necessarily be correct. For example, while WebAssembly's validation prevents functions from being called with extra arguments, specific type information about the contents of a tuple or struct are lost since WebAssembly lacks recursive types. We discuss the consequences of this further in the methodology and conclusion sections.

Broadly speaking, there are three overarching pillars of safety in our compiler. First, through implementing our compiler in Rust, we are able to avoid a plethora of memory safety errors via the language's novel borrow checking system. Second, (the main thrust of our effort) by adding robust type checking to parsing and transformation passes, we avoid multiple kinds of errors that can arise during compilation. Finally, WebAssembly provides rudimentary semantic validation built into the language, in a way that is unprecedented in current machine-level assembly languages and which provides avenues for extending type safety future work.

3. Review of Literature

3.1. Rust

Rust is a systems programming language that is built with a static type system that provides strong guarantees about isolation, concurrency, and memory safety [7]. Static typing ensures that programs written in Rust are checked during compilation time for type safety (and consequently eliminating the need to perform such checks during runtime)[1]. In particular, while ordinary type systems provide primitives for ensuring that the composition and usage of types are *semantically* valid (such as for the use of abstract data types (ADTs) and higher order functions), Rust's type system goes further by guaranteeing that code written will be *temporally* and *spatially* valid in memory. That is, code compiled by Rust is guaranteed to be free of memory errors such as dangling pointers or double frees, as well as data races.

To achieve this form of type soundness, Rust utilizes an ownership-based memory system which specifies that all memory-allocated objects must have a unique owner on the stack. References to owned objects can then be *borrowed* in a way that temporarily allows the data to be used by other data structures or functions. Such references can either be given mutably or immutably, depending on whether write access is needed. We illustrate this with an code snippet adapted from Rust by Example [8]:

[1] Rust does offer some language constructs such as Rc and RefCell that allow Rust's ownership type checking to be deferred from compile time to runtime, however these are considered less idiomatic and should be avoided, if possible.

```
 1 let x = 5;
 2 let y = x;
 3 println!("x is {}, y is {}", x, y);  // prints "x is 5, y is 5"
 4
 5 let a = Box::new(5);
 6 println!("a contains: {}", a);       // prints "a contains: 5"
 7
 8 let b = a;
 9 // println!("a contains: {}", a);
10
11 temporarily_use_box(&b);
12 println!("b contains: {}", b);       // prints "b contains: 5"
13
14 destroy_box(b);
15 // println!("b contains: {}", b);
```

In line 1, we see a stack-allocated integer assigned to `x`, and in line 2, the value is copied to `y`, requiring no resources to be moved. Both values can be used independently, as demonstrated in line 3.

In line 5, we see `Box` is used to explicitly allocate an integer to the heap; the pointer is stored in `a`. In line 8, since the integer value is on the heap, only the pointer address of `a` gets copied to `b`. Thus in Rust's type system, we say that the value of `a` gets moved to `b` – meaning that `b` now has ownership of the integer. Thus, trying accessing the data from `a` as suggested in line 9 would cause an error.

In line 11, we see a function (definition omitted) called that takes an immutable reference to `b`. Ownership is retained by `b`, so the value can still be used later, as seen in line 12. However, in line 14 a different function is called that takes ownership of `b`'s value. Thus, attempting to access `b`'s data as suggested in line 15 would cause an error.

Within this ownership model, several invariants are maintained to maximize programming flexibility while guaranteeing memory safety. For example, multiple immutable references of an object can be created and used, while only a single mutable reference of an object can be used at a time. This mirrors the notion of a readers-writer lock in concurrent programming. Likewise, references cannot live past the lexical scope of the original object (as to prevent dangling pointers). This invariant is maintained through *lifetimes*, a notion which associates lexical bounds with the types of individual objects and references. Through the use of lifetimes as a way to track object permanence during the type checking phase of Rust's compiler, the language eliminates any need for garbage collection. Instead, the compiler automatically determines which objects can be destructed and deallocated based on when they go out of scope.

While Rust was designed with a goal of robust type safety, it was not built from the ground-up with a formally proven specification [7]. However, several efforts have since been made to fill in this gap. *Oxide: The Essence of Rust* is one recent paper that discusses Rust's type system in depth, as part of a dissection of Rust's language through a formal semantics (which includes a complete set of type judgement rules that simulate a subset of modern Rust) [9]. In the context of the broader Rust ecosystem, this validates that even though many language primitives such as vectors are internally implemented in a type-unsafe way, the external APIs they provide allow for the basis of a safe type system for the rest of Rust's standard libraries to be built upon. The authors demonstrate this by modeling Rust's type system using a set of "places environments" and "provenances" that variables and references refer to, through which Rust's borrow checker (the part of the compiler which validates ownership) can successfully ensure that there are no use-after-free errors for references.

3.2. WebAssembly

WebAssembly, as defined in the WebAssembly Core Specification, is a low-level, assembly-like language designed for executing code efficiently while also maintaining a compact and portable representation [6]. The language is designed to be fast, safe, well-defined, hardware-independent, language-independent, platform-independent, open, modular, efficient, and easy to inspect and debug. It is also designed to be streamable and parallelizable, in that decoding, validation, and compilation of WebAssembly binaries can be split into parallel tasks and begin before all data has been seen (e.g.

when a file is downloaded from a web server)[2]. It achieves these goals through a virtual instruction set architecture which is based in a stack-machine-based computational model.

Executing WebAssembly code in a runtime environment (such as a browser) occurs in three stages: decoding, validation, and execution. During decoding, WebAssembly modules that are imported in a binary format are converted into an internal representation. Decoded modules are then validated to guarantee that they are safe. Validation ensures that local and global variables only contain type-correct values, instructions are only applied to operands of the expected types, and function invocations always evaluate to results of the right types. It also ensures, in conjunction with the language's semantics, that no memory location will be accessed except for those explicitly defined in the program, and that there is no undefined behavior (outside of traps or divergence). Upon successful validation, the module gets executed in two phases: instantiation, whereby a dynamic representation of the module is loaded into memory (including all necessary imports), and invocation, whereby functions may be invoked, and the results are returned to the host of the WebAssembly environment [6].

In order to support large data structures within its stack-machine-based instruction model, WebAssembly uses linear memories. A linear memory is a contiguous region of memory where values can be stored and access during execution. These linear memories can serve like heaps, except that the amount of memory intended to be used must be specified by the module ahead of time, removing the need for allocations and deallocations during runtime[3]. Specific values can be loaded from memory through fine-grained alignment and offset parameters in instructions.

[2] There are also plans to support parallelism in the form of multi-threaded computation through an upcoming proposal [10].
[3] Only one linear memory can be defined or imported per module in the current specification of WebAssembly, but this restriction may be lifted in future versions to allow for more granular memory management [6].

The most significant utility of WebAssembly today is that it serves as a tool to defer performance-sensitive code off of JavaScript for Web applications. The execution of JavaScript, broadly speaking, requires several more phases to execute than WebAssembly, including rapid code optimization (and de-optimization), as well as garbage collection, both of which restrict the optimal speed of execution in best-case scenarios [11]. Beyond this, WebAssembly is also significantly more compact than JavaScript since it has a binary format (alongside an interchangeable text format that can be used for debugging purposes). For these reasons, WebAssembly is a highly relevant compilation target for the design of a compiler.

We choose WebAssembly over other well-known compilation targets such as LLVM for several reasons. While LLVM provides both intermediate representations (IRs) and binary formats that could feasibly be used and extended, the IR is generally not portable (as a single program will typically have different representations for different machine architectures), which could make it more difficult to demonstrate that a compiler is type safe from end-to-end in future work. Secondly, while LLVM's compiler toolchain provides a host of compiler optimizations that could easily be applied for the sake of performance testing, this comes with the cost that the IR has large gaps of undefined behavior that are not ideal for the purpose of designing a compiler that is type-safe from end-to-end [12]. Finally, WebAssembly has a simpler instruction set architecture which makes it ideal for the subject of a research project.

4. Methodology

Discussion of the exact compiler implementation is separated into eight sections. Section 1 introduces the source language, and provides an introduction to existential types. Section 2 provides an overview of the compiler's multi-pass architecture. Section 3 introduces parsing and type checking. Section 4 explains the process of closure conversion. Section 5 introduces lambda lifting and record elimination. Section 6 examines code generation. Sections 7 and 8 briefly discuss our testing process and libraries that were used during development.

4.1. Source Language

The source language of our compiler is a typed variation of Scheme that has been customized for ease of readability and parsing, and whose S-expression syntax emphasizes the functional aspects of the language that are shared with other LISP dialects such as Racket and Closure. The syntax has been defined in Figure 1.

Our language features primitive datatypes such as integers, booleans, and strings, as well as complex datatypes such as tuples, records, and homogenous lists. One key difference from traditional LISP dialects we wish to highlight are the inclusion of explicit type annotations for lambda expressions. For example, a function that doubles its (integer) argument would be written as follows:

```
(lambda ((n : int)) : int (* x 2))
```

The consequence of type annotations is that all expressions in our language can be statically type-checked, so no type inference is required.

In order to support type checking during closure conversion, we extend our language with additional constructs to support existential types. Annotating a value with an existential type indicates that functions and variables outside of it cannot know (i.e. depend on) the actual internal representation, so it can only be referred to opaquely using a type variable.

Existential typing in our language are modeled after the presentation given in Types and Programming Languages [3]. We introduce a `pack` expression, which joins an expression with its substitution type, along with the explicit existential type annotation. Our type checker correctly validates that when the existential type is substituted with the provided type substitution, it yields back the original type. Likewise, we also introduce an `unpack` expression, which takes an existentially typed expression and assigns it to a bound identifier for use in a body expression (similar to `let`). In this

Syntax

Types	t := int	*Integer*
	| bool	*Boolean*
	| str	*String*
	| (list t_1)	*List (homogenous)*
	| (tuple t_1 ... t_n)	*Tuple*
	| (record (field$_1$: t_1) ... (field$_\text{n}$: t_n))	*Record*
	| (-> t_1 ... t_n)	*Function*
	| T$_0$, ... T$_\text{n}$	*Type variables**
	| (∃ T$_\text{n}$ t_1)	*Existential type**
Expressions	e := *num*	*Integer*
	| *bool*	*String*
	| *str*	*Boolean*
	| (+ e_1 e_2) | (- e_1 e_2) | (* e_1 e_2) | (/ e_1 e_2)	*Arithmetic ops*
	| (and e_1 e_2) | (or e_1 e_2)	*Boolean ops*
	| (< e_1 e_2) | (= e_1 e_2) | (> e_1 e_2)	*Comparison ops*
	| (concat e_1 e_2)	*Concatenation*
	| (cons e_1 e_2)	*Construct list*
	| (car e_1)	*Head of list*
	| (cdr e_1)	*Tail of list*
	| (null t_1)	*Empty list*
	| (null? e_1)	*Test for empty list*
	| (make-tuple e_1 ... e_n)	*Construct tuple*
	| (tuple-ref e_1 *num*)	*Tuple accessor*
	| (make-record (field$_1$ e_1) ... (field$_\text{n}$ e_n))	*Construct record*
	| (record-ref e_1 field)	*Record accessor*
	| (if e_1 e_2 e_3)	*Conditional*
	| (let ((id$_1$ e_1) ... (id$_\text{n}$ e_n)) e_body)	*Define local vars*
	| (set! id$_1$ e_1)	*Variable assignment*
	| (begin e_1 ... e_n)	*Sequential evaluation*
	| (lambda ((p$_1$: t_1) ... (p$_\text{n}$: t_n)): t_ret e_body)	*Anonymous function*
	| (pack t_sub e_package as t_exist)	*Construct package**
	| (unpack (T$_\text{n}$ e_package as name) in e$_\text{body}$)	*Unpack package**

Figure 1: The syntax of our typed Scheme. Types and expressions annotated with (*) are not allowed in the source language, but can be generated during intermediate compiler transformations for the purpose of typed closure conversion.

case, our type checker validates that the body is well typed (and has no free type variables) given the additional type variable provided by `unpack`.

For example, suppose we have the following existentially-typed expression assigned to the identifier p:

```
(pack (make-record (a 0)
                   (f (lambda ((x : int)) : int (+ 1 x))))
      int as
      (exists T_0 (record (a : T_0) (f : (-> T_0 int)))))
```

From the outside, this should be interpreted based on the last line as "a record of two fields, whose first field has type T_0, and whose second field is a function that takes type T_0 and produces an integer." The implementation uses `int` as the hidden type — observe that substituting `int` into the existential type annotation produces the actual type of the record defined in the first two lines. Since the inner type is hidden, functions from the outer scope cannot apply the function inside the record to any arbitrary value, since substituting a concrete type in place of T_0 is invalid. However, since the record's `a` field has a value of type T_0, it can be applied as shown (yielding the value 1):

```
(unpack (T_1 p as q) in ((record-ref q f) (record-ref q a)))
```

The key purpose of existential types in our compiler is to provide a means of abstracting away the type information of certain program code. Specifically, in our compiled code, environments that get passed into lambda expressions after closure conversion are given existential types. This is necessary in order to hide details about the environment's internal type representation from program code at the lambda's calling sites. Through existential typing, we guarantee that the only environment that can be passed into a lambda is the one that was constructed for it during closure conversion (even though there may be other environments available within the lambda's lexical scope that share the same internal type).

```
struct Expr {
    kind: Box<ExprKind<Expr>>,
}

struct TypedExpr {
    typ: Type,
    kind: Box<ExprKind<TypedExpr>>,
}

enum ExprKind<E> {  // where E is Expr or TypedExpr
    Num(i32),
    Bool(bool),
    Str(String),
    If(E, E, E),
    Let(Vector<(String, E)>, E),
    Lambda(Vector<(String, Type)>, Type, E),
    ...
}
```

Figure 2: The parameterized, mutually recursive data structures used by the compiler to represent abstract syntax trees throughout all compiler passes.

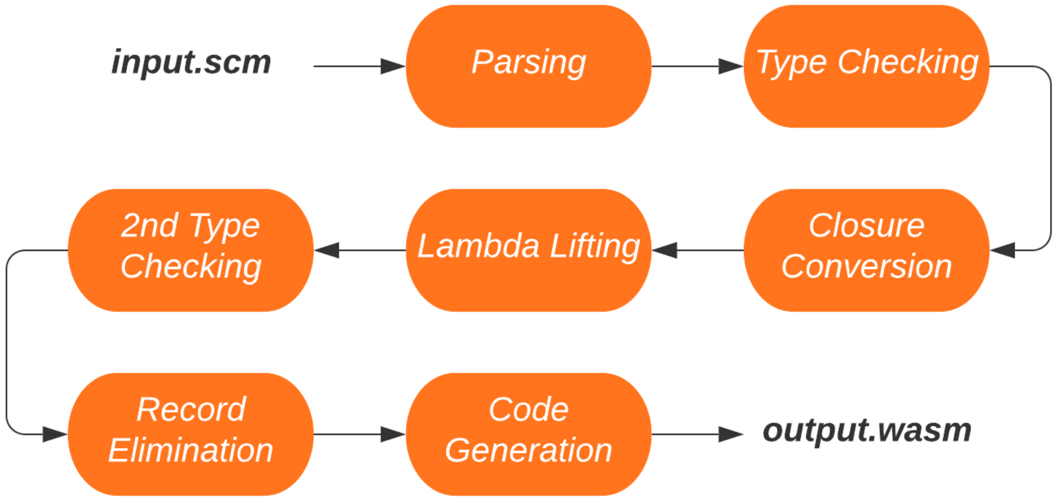
4.2. Compiler Architecture

Our compiler is implemented in Rust using a set of multiple compiler passes. Throughout compilation, the core data structure the passes operate on are abstract syntax trees (ASTs). For ease of implementation and to avoid potential errors grounded in mutation, we implement ASTs as immutable Rust enums. These ASTs come in two flavors, typed and untyped. Typed ASTs assign each node in the AST a type annotation, while untyped ASTs do not. We implement these as distinct data structures (opposed to using an optional field, for example) in order to minimize the potential error handling and to clearly distinguish when individual compiler passes rely on type information or not. A rough sketch of this code is provided in Figure 2.

We highlight this in order to emphasize the fact that delineating AST nodes into explicit cases based on the kind of expression allows us to develop tools for manipulating

ASTs that result in more maintainable code. While the Rust language makes it more challenging to manipulate S-expressions when compared to Lisp dialects such as Racket (which provide tools like quoting and quasiquoting), in implementing our compiler we have designed several AST meta-transformation functions that allow us to eliminate redundant transformation code for expression kinds that are unchanged within an individual compiler pass. This is akin to providing an interface which simply needs to be overwritten for the specific expression kinds that a compiler pass needs to transform.

We outline the compiler passes used as illustrated:



4.3. Parsing and type checking

We begin by parsing the user's input code, which is provided as a string. This ensures that the code is well-formed syntactically. As a result of the internal representation of our syntax trees, this will already reject code that violates rudimentary properties, such as not including the correct number of arguments to built-in functions, or including a type where an expression is to be expected.

Following this, we type check the user's code in order to validate that the program is well-typed. In our current implementation, closure conversion takes in an untyped AST since the result needs to be completely re-annotated with types afterwards. Hence, the type annotations generated from the initial type checking phase will get discarded. However, we still recommend performing this step in order to weed out potential typing errors before the compiler begins performing AST transformations.

As a running example, we will consider compiling the following program:

```
((lambda ((x : int)) : bool (< x 5)) 3)
```

In this program, we have defined a lambda expression which returns true if and only if its input is less than five. This function then gets applied to the value 3. During parsing, this program (in string format) is converted into an AST within Rust, and during type checking, we check that the types applied are correct. The lambda expression is assigned the type `(-> int bool)`, and 3 is assigned the type `int`, so the combined function application is valid and has type `int`.

4.4. Closure conversion

During closure conversion, our goal is to transform all lambda expressions in a way such that there are no more free variables within the lambda's body. By doing so, we can extract out lambda expressions to the top-level so that they can behave like global functions. (Observe that doing so while there are still free variables in the lambda bodies could result in ill-defined behavior). We achieve this goal by a) adding a new environment parameter to the beginning of each lambda's parameter list, b) transforming references to free variables with references into the environment parameter, c) pairing each lambda expression with a satisfying environment, and d) adjusting function applications of the lambda expressions to include the necessary environment as the first argument.

We use the pack expression to wrap the lambda-and-environment tuple (known as a closure) in an existential type, and we wrap the invocation sites with the unpack expression to allow the contents of the existentially typed closure to be used. This guarantees (through the application of our type checker) that at calling sites of the lambda, only the specific environment argument constructed for it can be passed in its environment parameter.

In our running example, the result of closure conversion is an AST of the following program:

```
(unpack (T₃ (pack (make-tuple
                    (lambda ((env1 : (record)) (x : int)) : bool (< x 5))
                    (make-record))
                  (record) as
                  (exists T₂ (tuple (-> T₂ int bool) T₂)))
         as temp0) in
    ((tuple-ref temp0 0) (tuple-ref temp0 1) 5))
```

Observe that since the body of our lambda expression contains no free variables, we end up with an empty environment being constructed as part of the closure.

4.5. Lambda lifting and record elimination

After the program has been closure converted, the program is lambda lifted. As described before, this compiler pass lifts all of the lambdas from their locations within the main program to top-level global definitions, substituting them with identifiers that refer to newly named functions in the global scope. Following this, the program is re-type checked to ensure that all references to function types (such as type annotations of higher order functions) have been updated to appropriate existential types.

The last transformation we perform before the ultimate code generation is record elimination. Recall that during closure conversion, all closure environments (which represent mappings from free variable names to their values) are constructed as records. Furthermore, the user may have constructed and used records within their own code. However, to simplify the implementation of code generation (discussed in the next paragraph), we choose to convert all record instances and types into corresponding tuple instances and types. We achieve this by observing that for any given record, the fields given are alphanumeric names, and hence can be sorted to define a unique and consistent order for the record values. Thus, we can simply permute the values in order corresponding to the sorted fields and discard the field names in order to obtain a type-safe equivalent representation as a tuple. Since information about field names is included even at the type annotation level, we can perform this transformation even in cases when there are no concrete instances of a record's type in the user's code.

In our running example, performing lambda lifting and record elimination yields the following program, consisting of a lifted function and main body expression:

```
funcs = { temp4: (lambda ((env1 : (tuple)) (x : int)) : bool (< x 5)) }

main = (unpack (T3 (pack (make-tuple temp4 (make-tuple))
                         (tuple) as
                         (exists T2 (tuple (-> T2 int bool) T2)))
                as temp0) in
           ((tuple-ref temp0 0) (tuple-ref temp0 1) 5))
```

4.6. Code generation

Finally, we transform the typed program into WebAssembly code. At a high level, code generation is performed in a similar manner as the AST-to-AST compiler passes, in which we recursively traverse the AST and dispatch based on the expression kind of each node. However, code generation poses several unique challenges, since we must generate different WebAssembly structures depending on whether we encounter primitive data types or control structures, complex data types, or functions.

Booleans, integers, conditionals, variable assignment, and operations such as arithmetic, boolean, and logical operations can all be represented and executed directly through corresponding stack-based instructions in WebAssembly. To minimize the implementation complexity, we represent all datatypes in WebAssembly using 32-bit integers, including pointers that are needed for more complex data types.

Complex data types such as tuples and lists are stored using WebAssembly's linear memory (see Section 3.2), and are represented on the stack using integer pointers. We use bump allocation in order to keep track of freely available memory for allocating data structures in our artificially constructed heap. A length n tuple gets translated by first constructing each of the tuple components in order (which themselves may require memory allocations), and then storing the n produced stack values as a sequence of n 32-bit values in linear memory. Likewise, lists are stored through a head-and-tail representation, where the first element of each pair is a value, and the second is either a pointer to the next node/value, or a sentinel indicating the null value [13]. By choosing these data structure representations, data access essentially falls out for free. Tuple projection, as well as car-ing or cdr-ing into lists[4], both correspond to indexing into memory with appropriate offsets.

Let expressions also pose a need to store values that may need to be accessed out of order. However, since newly bound variables need only to store the 32-bit data values and not the entire data structures, we can instead make use of a special WebAssembly feature known as local variables instead of maintaining local variables through the heap. In the context of WebAssembly, local variables allow a function to specify additional variable slots which can be accessed and reassigned to at any time within the function's body. By default, any lambda parameters are reserved to the prefix of an array named `locals` (so the nth argument passed to the function would be stored in `locals[n]`), and afterwards custom locals are assigned in order. The advantage of this design is that as a compiler designer, we are no longer burdened with the need to optimize register allocation, since this will be handled by WebAssembly.

The only challenges remaining are code generation of functions and existentially typed expressions. Since existentially typed expressions (constructed using pack) always encapsulate well-typed immutable closures during our compilation, we can simply access their inner contents directly while ignoring their extra type information. Similarly, unpack expressions can be handled like ordinary let expressions.

Lambda expression ASTs must be compiled into individual functions within the WebAssembly module. Each lambda parameter is mapped to a corresponding 32-bit

[4] LISP terminology for accessing the head or tail of a list.

placeholder within the WebAssembly function's parameters. To call the constructed functions, we must maintain a table during compilation which maps function names to indices, which allows us to perform function lookups whenever a function application occurs. While we did not include code generation of recursive functions due to time constraints, their definitions can be safely type-checked. The program's main expression is treated as its own zero-argument function, and we assign its function index to the *start* value in the WebAssembly module so it gets executed by default.

Our running example, after code generation, translates into the following WebAssembly module (shown in text format):

```
(module
  (type (;0;) (func (result i32)))
  (type (;2;) (func (param i32 i32)
        (result i32)))

  (func (;0;) (type 2) (param i32 i32)
        (result i32)
    (local i32 i32)
    local.get 1
    i32.const 5
    i32.lt_s)

  (func (;1;) (type 0) (result i32)
    (local i32)
    i32.const 0
    i32.const 0
    i32.const 0
    ...
    ...
    i32.const 0
    i32.store offset=4 align=1
    i32.store align=1
    i32.const 0
    local.set 0
    local.get 0
    i32.load offset=4 align=1
    i32.const 3
    local.get 0
    i32.load align=1
    call_indirect (type 2))

  (table (;0;) 32 funcref)
  (memory (;0;) 32)
  (export "$$MAIN$$" (func 1))
  (elem (;0;) (i32.const 0) func 0))
```

We can see that two functions are constructed – a two-argument function (corresponding to the lambda expression) that takes two i32 arguments representing the environment and integer respectively, as well as a zero-argument function (corresponding to the main expression). The module also declares several other pieces of module metadata, such as a list of the function types used in the module, a function lookup table, a linear memory module, an export declaration for the main function, and an entry for the function lookup table.

4.7. Testing

We utilize unit testing to individually validate each of the components of the compiler. In particular, we have over 100 unit tests that validate the type checker on a variety of source and intermediate programs, both correctly and incorrectly typed, that could arise at different stages of the compiler. We also have over 80 unit tests that validate the outputs produced by other passes in the compiler, and over 40 tests that validate compiled program output from end-to-end.

In particular, we highlight that we have tested end-to-end compilation of nested data structures (such as tuples of tuples, lists of tuples, tuples of lists of tuples, etc.), multi-argument functions, curried functions, functions that define local state (such as `make-adder`[5]), list operations (such as prepend), and functions that accept function inputs (such as non-polymorphic `apply`).

4.8. Libraries

In order to implement this compiler in Rust, we make use of several different libraries from crates.io, Rust's library ecosystem, in an effort to ensure our code is both maintainable and idiomatic.

For parsing source code into abstract syntax trees (ASTs), we utilize the `lexpr` crate, which provides utilities for parsing, printing, and manipulating S-expression data. Concretely speaking, `lexpr` performs both lexical analysis and semantic parsing of the input strings into a generic S-expression data structure, which we then convert using pattern matching into our individualized (untyped) AST data structure.

For consistency in maintaining ASTs as completely immutable structures (that require cloning in order to mutate), we use the immutable vectors provided by the `im` crate wherever we need to define a vector of items in our trees.

[5] `make-adder` is a higher-order function that takes an integer argument `x`, and returns a pure function that adds `x` to its input.

For generating the output WebAssembly code, we use the `parity-wasm` crate which provides utilities for constructing WebAssembly functions and modules through a collection of fluent interface based constructors. `parity-wasm` also handles the process of converting said modules into exportable binaries. In this way, we are able to abstract away from having to perform several bookkeeping tasks necessary for formatting code in either the WebAssembly text or binary formats, but we still are able to maintain full control over the instructions, type signatures, local variables, and so forth that are required to define the functionality of a WebAssembly module.

Lastly, we also make use of `wasmer-runtime`, a crate that allows us to execute WebAssembly code generated by `parity-wasm` for the purpose of unit testing during the Rust runtime.

5. Results

Our main result is that by developing a compiler with type-safe transformations and by creating a type-checker which is compatible with all intermediate code representations, we are able to 1) derive several benefits during the process of building and maintaining the compiler, and 2) argue there are further advantages when one tries to extend the features of a language or optimize the code a compiler produces.

Firstly, we argue that our type-checker is robust. Through Rust's pattern matching features, we can guarantee at compile time that the type checker has handles every expression kind defined in the source language. Since our language is mostly functional in nature (with the exception of the set! construct), we are able to safely analyze large classes of programs that omit mutation simply through static analysis[6]. We utilize type environments that are passed through recursive type-checking calls to

[6] Lists, tuples, and records in our language are all immutable. One caveat to the model of mutation we do permit is that assignment of variables from outer scopes will not work after closure conversion is performed, so we must restrict `set!` to only operate on variables local to the closest lambda expression.

ensure that bound variables can be used only in the contexts in which they are defined (e.g. a local variable cannot be used outside of the `let` expression it was defined in).

We also make use of appropriate code abstractions within the compiler to ensure the type checking is uniform, e.g. all lists of values are type checked in the same way. Over 100 unit tests are used to validate appropriate type checking behavior (both passing and failing) for catching errors such as empty `begin` expressions, out-of-bounds tuple referencing, incorrect argument arities, non-matching types for control flow branches, and so on. Special attention is given to type checking edge cases for `pack` and `unpack` expressions (based on the treatment given in Types and Programming Languages [3]), despite the fact that many will not pop up since only certain forms of these expressions are generated by closure conversion. Due to time constraints, we omitted checking for identifier reuse in variable, parameter, or record field names, so our compiler assumes that all source programs do not use variable shadowing or duplicate field names within records.

When building individual compiler passes, the type checker can catch a wide variety of bugs. For example, when implementing a transformation which changes one type to another, there are often dozens of locations in the code where we must insert function calls to perform the appropriate conversion method. However, there were several occasions during the development of our compiler when a location was accidentally not updated, leading to bugs. For example, in one case record elimination was not taking place on a lambda's parameter and return types[7]. This error was caught by noticing that type checking failed on a compiled lambda expression with record arguments. This failure led us to noticing the discrepancy between the types of the actual record (which was converted into a tuple) and the record parameter (which was not being updated as intended).

[7] See https://github.com/Chriscbr/scheme-to-wasm/commit/bfa5e61a5013bed259f9a00d8aeeb944c39f9d5f.

We also argue that similar benefits can occur when making modifications to the compiler. For example, when adding a new built-in construct (such as `incr`) to our language, we would notice two immediate benefits over trying to make such a change in a compiler that is not type-safe and which is written in a less type-safe language such as C or C++. Since our compiler is written in Rust, we benefit from use of its type-safe pattern matching feature. This essentially implies that after adding a new kind of expression, the Rust compiler will produces an error for each place in the code that needs a new case to handle `incr`. But furthermore, we have a type-checker which runs in between many of the compiler passes. Thus as long we correctly implement type checking for `incr`, we can simply write end-to-end test cases using the new code construct, and the compiler will automatically allow us to discover in which stage of the compiler there are bugs in the implementation, if any. The key takeaways are that using the type checker allows us to discover bugs easier, and without needing to run the code through all compiler passes.

6. Future Work

We believe there are several avenues for future work. In terms of the base compiler implementation, we have left out code generation of strings and string-based operations. We also believe it would be useful to try extending the language with other programming language constructs, such as recursive functions and polymorphic types, in order to analyze the difficulties of a) designing type-safe compiler passes for these constructs, and b) implementing code generation of these constructs targeting WebAssembly. It is also important to research if this kind of compiler safety can be maintained if we extend our language to use garbage collection.

There are also opportunities to extend this work by improving the type-checking capabilities to code generation, the final compiler pass. One possible avenue for attacking this is to extend the WebAssembly language with additional type annotations that include more fine-grained parameter and return types of functions based on the types in our source language. Such a method could be used to extend WebAssembly with existential type annotations imitating the type information used in Typed Assembly Language, to guarantee that WebAssembly functions only get called with environment arguments that belong to their respective closures [14].

7. Conclusion

We have developed a fully operational compiler which translates code from a high-level functional language to a low level assembly language. Our compiler augments the level of type safety beyond the level provided in traditional compilers by making repeated use of a robust type-checker which can be used to validate the output of all AST transformations. By doing so, we find evidence that we can debug, maintain, and extend our compiler much easier through the guarantees that type-safety provide in between compiler passes.

8. References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with WebAssembly,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, Barcelona, Spain, 2017, pp. 185–200, doi: 10.1145/3062341.3062363.

[2] “About Emscripten — Emscripten 1.39.13 documentation.” https://emscripten.org/docs/introducing_emscripten/about_emscripten.html (accessed Apr. 21, 2020).

[3] B. C. Pierce, *Types and programming languages*. Cambridge, Mass: MIT Press, 2002.

[4] *World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation*. 2019.

[5] C. Eberhardt, *The future of WebAssembly - A look at upcoming features and proposals*. 2018.

[6] A. Rossberg, *WebAssembly Specification — WebAssembly 1.0*.

[7] N. D. Matsakis and F. S. Klock, “The rust language,” *ACM SIGAda Ada Lett.*, vol. 34, no. 3, pp. 103–104, Nov. 2014, doi: 10.1145/2692956.2663188.

[8] “Ownership and moves - Rust By Example.” https://doc.rust-lang.org/rust-by-example/scope/move.html (accessed Apr. 23, 2020).

[9] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed, “Oxide: The Essence of Rust,” *ArXiv190300982 Cs*, Mar. 2019, Accessed: Apr. 20, 2020. Available: http://arxiv.org/abs/1903.00982.

[10] *WebAssembly/threads*. WebAssembly, 2020.

[11] L. Clark, *What makes WebAssembly fast? – Mozilla Hacks - the Web developer blog*. .

[12] “FAQ - WebAssembly.” https://webassembly.org/docs/faq/ (accessed Apr. 22, 2020).

[13] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and interpretation of computer programs*, 2. ed. Cambridge, Mass.: MIT Press, 1996.

[14] G. Morrisett, D. Walker, K. Crary, and N. Glew, “From system F to typed assembly language,” *ACM Trans. Program. Lang. Syst. TOPLAS*, vol. 21, no. 3, pp. 527–568, May 1999, doi: 10.1145/319301.319345.