Compiling Quantitative Type Theory to Michelson for Compile-Time Verification and Run-time Efficiency in Juvix

Christopher Goes(✉)

Metastate AG, Zug, Switzerland
cwgoes@metastate.dev

Abstract. Michelson, the stack-based virtual machine of the Tezos blockchain, integrates type-checking for program execution completion but not program correctness. Manual stack tracking is efficient but less ergonomic to write in than a higher-level lambda calculus with variables. Compiling McBride's Quantitative Type Theory to Michelson allows for compile-time verification of semantic predicates and automatic stack optimisation by virtue of the type-theoretic usage accounting system.

Keywords: qtt · Michelson · Tezos · Juvix

1 Introduction and Prior Work

Smart contracts running on distributed ledgers are an archetypal example of a security-critical application, and one where the popular conceit of security-through-obscurity cannot serve since contract code is public, yet results so far from languages such as Solidity [1], the contract language most popular on the Ethereum blockchain [2], have not been promising. Numerous hacks and losses numbering in the hundreds of millions [3,4] have resulted from often quite simple bugs in contracts.

Michelson is the smart contract language of Tezos [5]. The Michelson language is stack-based, with high-level data types & primitive functions. A Michelson program consists of a series of instructions, each of which describes a state transition rule which re-writes the stack. At compile-time, static type-checking ensures that the instruction sequence has the correct stack types by starting with the initial stack type & walking through the start & return stack types of each instruction. An run-time, instructions are executed in sequence according to the rewrite rules, with input values provided by the caller (e.g. as arguments to a smart contract call). Michelson's static type checking provides for verification of executability but not for verification of semantic correctness—it can say nothing about how the start & end storage values of a contract relate to each other or to the arguments provided in the contract call.

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12478, pp. 146–160, 2020.
https://doi.org/10.1007/978-3-030-61467-6_10

Due to its stack-based nature, pure Michelson is not particularly ergonomic to develop in—the contract author must mentally track which stack position corresponds to what value at each instruction in the sequence of instructions which together comprise the contract, and limited facilities exist for function abstraction and avoidance of code duplication. For this reason, prior efforts have build intermediate languages for Michelson, such as Albert [6], which allow the programmer to use a higher-level syntax to define functions which operate on named variables, and automatically handle the conversion to Michelson operation sequences by tracking the relationship between variables names & stack positions during compilation. Due to the higher-level abstraction, however, these techniques frequently come at a cost in runtime efficiency, since the automatic translation cannot easily take into account whether a variable used as an argument to a function will need to be used later (and so must be kept around on the stack) or is only used once (and so can be safely discarded), as a programmer might do mentally when writing low-level Michelson by hand.

Prior efforts to bring semantic verification to Michelson, in particular Nomadic Labs' *Mi-Cho-Coq* [7] framework for the verification of Michelson contracts in Coq, have provided verification capabilities by expressing the semantics of Michelson in an existing theorem prover. This allows for precise verification of contract behavioural semantics, but requires that such verification be done at the low-level of Michelson instruction sequences—so if a developer writes a contract in a higher-level language which compiles to Michelson, they will need to perform verification on the translated Michelson output, instead of in the higher-level language itself. Furthermore, the verification must operate at the level of Michelson stack semantics—one cannot, for example, express invariants about the behaviour of functions with named variables written in the higher-level language, but must instead reason about values at particular stack positions. Of course, this has the advantage of guarding against mistakes in the translation from the higher-level language to Michelson, but requires this verification overhead in every analysis performed—ideally, one would express properties of each unique contract in the semantics of the higher-level language, verify (once) the compiler transformation to the semantics of Michelson, and thereby obtain an equivalent level of assurance.

Outside the Tezos ecosystem, Edstrom & Pettersson's prior effort to realise dependently-typed smart contracts [8] achieved high-level semantic verification, but found the output code to be too inefficient to execute. They wrote an Idris [9] backend targeting Ethereum's LLL language [10]. Our approach shares a similar goal of high-level verification, but utilises a bespoke compilation pipeline—their approach handicapped itself by compiling to LLL instead of directly to EVM opcodes, and Idris' lack of linearity meant that they had to perform expensive memory-management operations in output contracts.

Conor McBride's Quantitative Type Theory (QTT) [11] elegantly melds full-spectrum dependent-type semantics with precise usage accounting. In our high-level smart contract language Juvix, we alter QTT to include the semantics of built-in Michelson operations in the higher-level language so that semantic verification can be performed in the language in which the developer is writing, and

we utilise the precise usage information to optimise the output code produced by our Michelson compilation pipeline. Juvix also includes a high-level frontend language, datatype system, pattern matching, etc., but those abstractions can be desugared to the core representation, so this paper describes only the core altered quantitative type theory & the variable-usage stack accounting used in the Michelson compilation pipeline. We expect that this fundamental approach could also be reused with other frontends or other stack-based virtual machines without much difficulty.

2 Core Language

Our core syntax & type theory is based on QTT, altered to include additional primitives, and instantiated over the semiring of the natural numbers plus ω for maximum granularity in expressing usage information.

2.1 Preliminaries

A *semiring* R is a set R with binary operations $+$ (addition) and $\cdot$ (multiplication), such that $(R, +)$ is a commutative monoid with identity 0, $(R, \cdot)$ is a monoid with identity 1, multiplication left and right distribute over addition, and multiplication by 0 annihilates R.

The core type theory must be instantiated over a particular semiring. Choices include the boolean semiring $(0, 1)$, the zero-one-many semiring $(0, 1, \omega)$, and the natural numbers with addition and multiplication.

We instantiate the type theory over the semiring of natural numbers plus ω, which is the most expressive option—terms can be 0-usage ("contemplated"), n-usage ("computed n times"), or ω-usage ("computed any number of times").

Let S be a set of sorts (i, j, k) with a total order.

Let K be the set of primitive types, C be the set of primitive constants, and $\vdots$ be the typing relation between primitive constants and primitive types, which must assign to each primitive constant a unique primitive type and usage. When instantiated for compiling to Michelson, these sets are the sets of built-in types & values in the Michelson language [12].

Let F be the set of primitive functions, where each f is related to a function type, including an argument usage annotation, by the $\vdots$ relation and endowed with a reduction operation ${}^{\rightarrow f}$, which provided an argument of the function input type computes an argument of the function output type. When instantiated for compiling to Michelson, this set is the set of built-in operations in the Michelson language, e.g. `ADD`, `MUL`, `NOT`, etc., endowed with appropriate types.

Primitive types, primitive constants, and primitive functions are threaded-through to the untyped lambda calculus to which the core language is erased, so they must be directly supported by the low-level execution model, in this case Michelson. The core type theory and subsequent compilation pathways are parameterised over K, C, F, $\vdots$, and the reduction operations ${}^{\rightarrow f}$, which are assumed to be available as implicit parameters.

2.2 Syntax

Our syntax is inspired by the bidirectional syntax of Conor McBride in I Got Plenty o' Nuttin' [13].

Let R, S, T, s, t be types & terms and d, e, f be eliminations, where types can be synthesised for eliminations but must be specified in advance for terms (Fig. 1).

$$
\begin{array}{lll}
R, S, T, s, t ::= *_i & & \text{sort } i \\
\quad \mid \kappa \in K & & \text{primitive type} \\
\quad \mid (x \overset{\pi}{:} S) \to T & & \text{function type} \\
\quad \mid (x \overset{\pi}{:} S) \otimes T & & \text{dependent multiplicative conjunction type} \\
\quad \mid \lambda x.t & & \text{abstraction} \\
\quad \mid e & & \text{elimination}
\end{array}
$$

$$
\begin{array}{lll}
d, e, f ::= x & & \text{variable} \\
\quad \mid c \in C & & \text{primitive constant} \\
\quad \mid f \in F & & \text{primitive function} \\
\quad \mid fs & & \text{application} \\
\quad \mid (s, t) & & \text{multiplicative conjunction} \\
\quad \mid let\ (x, y) = d\ in\ e & & \text{dependent multiplicative conjunction pattern match} \\
\quad \mid s \overset{\pi}{:} S & & \text{type \& usage annotation}
\end{array}
$$

Fig. 1. Core syntax

Sorts $*_i$ are explicitly levelled. Dependent function types, dependent conjunction types, and type annotations include a usage annotation π.

Judgements have the following form:

$$x_1 \overset{\rho_1}{:} S_1, \ldots, x_n \overset{\rho_n}{:} S_n \vdash M \overset{\sigma}{:} T$$

where $\rho_1 \ldots \rho_n$ are elements of the semiring and σ is either the 0 or 1 of the semiring.

Further define the syntactic categories of usages ρ, π and precontexts Γ:

$$\rho, \pi \in R$$
$$\Gamma := \diamond \mid \Gamma, x \overset{\rho}{:} S$$

The symbol $\diamond$ denotes the empty precontext.

Precontexts contain usage annotations ρ on constituent variables. Scaling a precontext, $\pi\Gamma$, is defined as follows:

$$\pi(\diamond) = \diamond$$
$$\pi(\Gamma, x \overset{\rho}{:} S) = \pi\Gamma, x \overset{\pi\rho}{:} S$$

Usage annotations in types are not affected.

By the definition of a semiring, 0Γ sets all usage annotations to 0.

Addition of two precontexts $\Gamma_1 + \Gamma_2$ is defined only when $0\Gamma_1 = 0\Gamma_2$:

$$\diamond + \diamond = \diamond$$
$$(\Gamma_1, x \overset{\rho_1}{:} S) + (\Gamma_2, x \overset{\rho_2}{:} S) = (\Gamma_1 + \Gamma_2), x \overset{\rho_1+\rho_2}{:} S$$

Contexts are identified within precontexts by the judgement $\Gamma \vdash$, defined by the following rules:

$$\frac{}{\diamond \vdash} \text{ Emp}$$

$$\frac{\Gamma \vdash \qquad 0\Gamma \vdash S}{\Gamma, x \overset{\rho}{:} S \vdash} \text{ Ext}$$

$0\Gamma \vdash S$ indicates that S is well-formed as a type in the context of 0Γ. *Emp*, for "empty", builds the empty context, and *Ext*, for "extend", extends a context Γ with a new variable x of type S and usage annotation ρ. All type formation rules yield judgements where all usage annotations in Γ are 0—that is to say, type formation requires no computational resources).

Term judgements have the form:

$$\Gamma \vdash M \overset{\sigma}{:} S$$

where $\sigma \in 0, 1$.

Primitive constant term judgements have the form:

$$\vdash M \overset{\gamma}{:} S$$

where γ is any element in the semiring.

A judgement with $\sigma = 0$ constructs a term with no computational content, while a judgement with $\sigma = 1$ constructs a term which will be computed with.

For example, consider the following judgement:

$$n \overset{0}{:} Nat, x \overset{1}{:} Fin(n) \vdash x \overset{\sigma}{:} Fin(n)$$

When $\sigma = 0$, the judgement expresses that the term can be typed:

$$n \overset{0}{:} Nat, x \overset{1}{:} Fin(n) \vdash x \overset{0}{:} Fin(n)$$

Because the final colon is annotated to zero, this represents contemplation, not computation. When type checking, n and x can appear arbitrary times.

Computational judgement:

$$n \overset{0}{:} Nat, x \overset{1}{:} Fin(n) \vdash x \overset{1}{:} Fin(n)$$

Because the final colon is annotated to one, during computation, n is used exactly 0 times, x is used exactly one time. x can also be annotated as ω, indicating that it can be used (computed with) an arbitrary number of times.

2.3 Typing Rules

2.3.1 Universe (Set Type)

Let S be a set of sorts i, j, k with a total order.

2.3.1.1 Formation Rule

$$\frac{0\Gamma \vdash \qquad i < j}{0\Gamma \vdash *_i \overset{0}{:} *_j} *$$

2.3.1.2 Introduction Rule

$$\frac{0\Gamma \vdash V \overset{0}{:} *_i \qquad 0\Gamma, x \overset{0}{:} V \vdash R \overset{0}{:} *_i}{\Gamma \vdash (x \overset{\pi}{:} V) \to R \overset{0}{:} *_i} *\text{-Pi}$$

Sorts can be contemplated (typed in the $\sigma = 0$ fragment) only.

2.3.2 Primitive Constants

2.3.2.1 Formation and Introduction Rule

$$\frac{c \in C \qquad \kappa \in K \qquad c \dot{:} (\gamma, \kappa)}{\vdash c \overset{\gamma}{:} \kappa} Prim - Const$$

Primitive constants are typed according to the primitive typing relation, and they can be produced in any computational quantity wherever desired.

2.3.3 Primitive Functions

2.3.3.1 Formation and Introduction Rule

$$\frac{f \in F \qquad f : (\gamma, (x \overset{\pi}{:} S) \to T)}{\vdash f \overset{\gamma}{:} (x \overset{\pi}{:} S) \to T} \text{Prim-Fn}$$

Primitive functions are typed according to the primitive typing relation, and they can be produced in any computational quantity wherever desired. Primitive functions can be dependently-typed—in the case of Michelson, polymorphic primitives such as `ADD` will be represented in the core language as dependently typed, i.e. `add : (t: Type) -> t -> t`, where `t` is restricted to the primitive types for which Michelson supports `ADD`.

2.3.3.2 Elimination Rule
Primitive functions use the same elimination rule as native lambda abstractions.

2.3.4 Dependent Function Types

Function types $(x \overset{\pi}{:} S) \to T$ record usage of the argument.

2.3.4.1 Formation Rule

$$\frac{0\Gamma \vdash S \qquad 0\Gamma, x \overset{0}{:} S \vdash T}{0\Gamma \vdash (x \overset{\pi}{:} S) \to T}\ \text{Pi}$$

2.3.4.2 Introduction Rule

$$\frac{\Gamma, x \overset{\sigma\pi}{:} S \vdash M \overset{\sigma}{:} T}{\Gamma \vdash \lambda x.M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T}\ \text{Lam}$$

The usage annotation π is not used in judgement of whether T is a well-formed type. It is used in the introduction and elimination rules to track how x is used, and how to multiply the resources required for the argument, respectively:

2.3.4.3 Elimination Rule

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T \qquad \Gamma_2 \vdash N \overset{\sigma'}{:} S \qquad 0\Gamma_1 = 0\Gamma_2 \qquad \sigma' = 0 \Leftrightarrow (\pi = 0 \vee \sigma = 0)}{\Gamma_1 + \pi\Gamma_2 \vdash MN \overset{\sigma}{:} T[x := N]}\ \text{App}$$

$0\Gamma_1 = 0\Gamma_2$ means that Γ_1 and Γ_2 have the same variables with the same types.

In the introduction rule, the abstracted variable x has usage $\sigma\pi$ so that non-computational production requires no computational input.

In the elimination rule, the resources required by the function and its argument, scaled to the amount required by the function, are summed.

The function argument N may be judged in the 0-use fragment of the system if and only if we are already in the 0-use fragment ($\sigma = 0$) or the function will not use the argument ($\pi = 0$).

2.3.5 Dependent Multiplicative Conjunction (Tensor Product)

Multiplicative conjunctions, colloquially referred to as "pair" type, can be dependent.

2.3.5.1 Formation Rule

$$\frac{0\Gamma \vdash A \qquad 0\Gamma, x \overset{0}{:} S \vdash T}{0\Gamma \vdash (x \overset{\pi}{:} S) \otimes T}\ \otimes$$

Type formation does not require any resources.

2.3.5.2 Introduction Rule

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} S \qquad \Gamma_2 \vdash N \overset{\sigma}{:} T[x := M] \qquad 0\Gamma_1 = 0\Gamma_2}{\pi\Gamma_1 + \Gamma_2 \vdash (M, N) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T}$$

This is similar to the introduction rule for dependent function types above.

2.3.5.3 Elimination Rules

$$\frac{\Gamma \vdash M \overset{0}{:} (x \overset{\pi}{:} S) \otimes T}{\Gamma \vdash fst_{\otimes}\ M \overset{0}{:} S}$$

$$\frac{\Gamma \vdash M \overset{0}{:} (x \overset{\pi}{:} S) \otimes T}{\Gamma \vdash snd_{\otimes}\ M \overset{0}{:} T[x := fst_{\otimes}(M)]}$$

Under the erased ($\sigma = 0$) part of the theory, projection operators can be used as normal.

$$\frac{0\Gamma_1, z \overset{0}{:} (x \overset{\pi}{:} S) \otimes T \vdash U \quad \Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \quad \Gamma_2, x \overset{\sigma\pi}{:} S, y \overset{\sigma}{:} T \vdash N \overset{\sigma}{:} U[z := (x, y)] \quad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash let\ (x, y) = M\ in\ N \overset{\sigma}{:} U[z := M]}\ \otimes \text{ Elim}$$

Under the resourceful part, both elements of the conjunction must be matched and consumed.

2.3.6 Variable and Conversion Rules

The variable rule selects an individual variable, type, and usage annotation from the context:

$$\frac{\vdash 0\Gamma, x \overset{\sigma}{:} S, 0\Gamma'}{0\Gamma, x \overset{\sigma}{:} S, 0\Gamma' \vdash x \overset{\sigma}{:} S}\ \text{Var}$$

The conversion rule allows conversion between judgmentally equal types:

$$\frac{\Gamma \vdash M \overset{\sigma}{:} S \qquad 0\Gamma \vdash S \equiv T}{\Gamma \vdash M \overset{\sigma}{:} T}\ \text{Conv}$$

Note that type equality is judged in a context with no resources.

2.3.7 Equality Judgements

Types are judgmentally equal under beta reduction:

$$\frac{\Gamma \vdash S \qquad \Gamma \vdash T \qquad S \to_{\beta} T}{\Gamma \vdash S \equiv T}\ \equiv\text{-Type}$$

Terms with the same type are judgmentally equal under beta reduction:

$$\frac{\Gamma \vdash M \overset{\sigma}{:} S \qquad \Gamma \vdash N \overset{\sigma}{:} S \qquad M \to_{\beta} N}{\Gamma \vdash M \equiv N \overset{\sigma}{:} S}\ \equiv\text{-Term}$$

As primitive types, values, and functions are included in the type theory, proofs about behavioural semantics can then be created in the usual fashion.

2.4 Erasure

Terms which are merely contemplated (in the $\sigma = 0$ fragment) are erased at compile-time, and thereby incur no runtime cost.

Define the core erasure operator $\blacktriangleright$.

Erasure judgements take the form $\Gamma \vdash t \overset{\sigma}{:} S \ \blacktriangleright \ u$ with $t \overset{\sigma}{:} S$ a core judgement and u an erased core term.

Computationally relevant terms are preserved, while terms which are only contemplated are erased.

Note that $\sigma / = 0$ must hold, as the erasure of a computationally irrelevant term is nothing.

2.4.1 Primitives and Lambda Terms

$$\frac{c \overset{\sigma}{:} S \qquad \sigma / = 0}{c \overset{\sigma}{:} S \ \blacktriangleright \ c} \text{ Prim-Const-Erase-+}$$

$$\frac{f \overset{\sigma}{:} S \qquad \sigma / = 0}{f \overset{\sigma}{:} S \ \blacktriangleright \ f} \text{ Prim-Fun-Erase-+}$$

$$\frac{\vdash 0\Gamma, x \overset{\sigma}{:} S, 0\Gamma' \qquad \sigma / = 0}{0\Gamma, x \overset{\sigma}{:} S, 0\Gamma' \vdash x \overset{\sigma}{:} S \ \blacktriangleright \ x} \text{ Var-Erase-+}$$

$$\frac{t \overset{\sigma}{:} T \blacktriangleright u \qquad \sigma\pi = 0}{\lambda x.t : (x \overset{\pi}{:} S) \to T \blacktriangleright u} \text{ Lam-Erase-0}$$

$$\frac{t \overset{\sigma}{:} T \blacktriangleright u \qquad \sigma\pi / = 0}{\lambda x.t : (x \overset{\pi}{:} S) \to T \blacktriangleright \lambda x.u} \text{ Lam-Erase-+}$$

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T \blacktriangleright u \qquad \Gamma_2 \vdash N \overset{0}{:} S \qquad \sigma\pi = 0}{\Gamma_1 \vdash MN \overset{\sigma}{:} T[x := N] \blacktriangleright u} \text{ App-Erase-0}$$

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} S) \to T \blacktriangleright u \qquad \Gamma_2 \vdash N \overset{\sigma\pi}{:} S \blacktriangleright v \qquad \sigma\pi / = 0}{\Gamma_1 + \Gamma_2 \vdash MN \overset{\sigma}{:} T[x := N] \blacktriangleright u\, v} \text{ App-Erase-+}$$

$$\frac{\Gamma \vdash s \overset{\pi}{:} S \qquad s \ \blacktriangleright \ u \qquad \pi / = 0}{\Gamma \vdash s \overset{\pi}{:} S \ \blacktriangleright \ u} \text{ Ann-Erase-+}$$

In the *Lam-Erase-0* rule, the variable x bound in t will not occur in the corresponding u, since it is bound with usage 0, with which it will remain regardless of how the context splits, so the rule *Var-Erase-+* cannot consume it.

2.4.2 Multiplicative Conjunction

2.4.2.1 Constructor

$$\frac{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \qquad \sigma / = 0 \qquad \pi / = 0 \qquad s \blacktriangleright u \qquad t \blacktriangleright v}{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \blacktriangleright (u,v)} \otimes\text{-Erase-++}$$

If the first element of the pair is used, the constructor is erased to the untyped constructor.

$$\frac{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \qquad \sigma / = 0 \qquad \pi = 0 \qquad t \blacktriangleright v}{\Gamma \vdash (s,t) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \blacktriangleright v} \otimes\text{-Erase-0+}$$

If the first element of the pair is not used, the constructor is erased completely.

2.4.2.2 Destructor

$$\frac{\Gamma_1 \vdash s \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \quad \Gamma_1 + \Gamma_2 \vdash let\ (x,y) = s\ in\ t \overset{\sigma'}{:} M[z := (x,y)] \quad \sigma, \sigma' / = 0 \quad s \blacktriangleright u \quad t \blacktriangleright v}{\Gamma_1 + \Gamma_2 \vdash let\ (x,y) = s\ in\ t \overset{\sigma'}{:} M[z := (x,y)] \blacktriangleright let\ (x,y) = u\ in\ v} \text{let-Erase-++}$$

If the pair is used, the destructor is erased to the untyped destructor.

$$\frac{\Gamma_1 \vdash s \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T \quad \Gamma_1 + \Gamma_2 \vdash let\ (x,y) = s\ in\ t \overset{\sigma'}{:} M[z := (x,y)] \quad \sigma = 0 \wedge \sigma' / = 0 \quad t \blacktriangleright v}{\Gamma_1 + \Gamma_2 \vdash let\ (x,y) = s\ in\ t \overset{\sigma'}{:} M[z := (x,y)] \blacktriangleright v} \text{let-Erase-0+}$$

If the pair is not used, the destructor is erased completely.

2.5 Reduction Semantics

Contraction is $(\lambda x.t : (\pi x : S) \to T)\ s \rightsquigarrow_\beta (t : T)[x := s : S]$.

De-annotation is $(t : T) \rightsquigarrow_\nu t$.

The reflexive transitive closure of $\rightsquigarrow_\beta$ and $\rightsquigarrow_\nu$ yields beta reduction $\to_\beta$ as usual.

2.5.1 Parallel-Step Reduction Let parallel reduction be $\triangleright$, operating on usage-erased terms, by mutual induction.

2.5.1.1 Basic Lambda Calculus

$$\frac{}{*_i \rhd *_i}$$

$$\frac{}{x \rhd x}$$

$$\frac{S \rhd S' \qquad T \rhd T'}{(x : S) \to T \rhd (x : S') \to T'}$$

$$\frac{t \rhd t'}{\lambda x.t \rhd \lambda x.t'}$$

$$\frac{f \rhd f' \qquad s \rhd s'}{fs \rhd f's'}$$

$$\frac{t \rhd t' \qquad T \rhd T'}{t : T \rhd t' : T'}$$

$$\frac{t \rhd t' \qquad S \rhd S' \qquad T \rhd T' \qquad s \rhd s'}{(\lambda x.t : (x : S) \to T)s \rhd (t' : T')[x := s' : S']}$$

2.5.1.2 Multiplicative Conjunction

$$\frac{S \rhd S' \qquad T \rhd T'}{(x : S) \otimes T \rhd (x : S') \otimes T'}$$

$$\frac{s \rhd s' \qquad t \rhd t'}{(s, t) \rhd (s', t')}$$

$$\frac{z \rhd (m, n) \qquad m \rhd m' \qquad n \rhd n' \qquad s \rhd s'}{let\ (x, y)\ =\ z\ in\ s \rhd s'[x := m', y := n']}$$

Reduction takes place inside a multiplicative conjunction.

2.5.1.3 Primitives

$$\frac{\kappa \in K}{\kappa \rhd \kappa}$$

$$\frac{c \in C}{c \rhd c}$$

Primitive types and primitive constants reduce to themselves.

$$\frac{f \in F \qquad x \rhd x' \qquad x' \to_f y}{fx \rhd y}$$

Primitive functions reduce according to the reduction operation defined for the function according to the Michelson semantics [12].

2.6 Examples

2.6.1 SKI Combinators

2.6.1.1 S Combinator The dependent S ("substitution") combinator can be typed as (Fig. 2):

$$\vdash \lambda t1.\lambda t2.\lambda t3.\lambda x.\lambda y.\lambda z.xz(yz) : (x : ((a : t1) \to (b : t2) \to t3)) \to (y : ((a : t1) \to t2)) \to (z : t1) \to t3$$

Fig. 2. S combinator

This will also typecheck if the x, y, and z argument usages are replaced with ω (instead of 1 and 2).

2.6.1.2 K Combinator The dependent K ("constant") combinator can be typed as (Fig. 3):

$$\vdash \lambda t1.\lambda t2.\lambda x.\lambda y.x : (t1 : *_i) \to (t2 : *_i) \to (x : t1) \to (y : t2) \to t1$$

Fig. 3. K combinator

This will also typecheck if the x and y argument usages are replaced with ω (instead of 1 and 0).

2.6.1.3 I Combinator The dependent I ("identity") combinator can be typed as (Fig. 4):

$$\vdash \lambda t.\lambda x.(x : t) : (t : *_i) \to (x : t) \to t$$

Fig. 4. I combinator

This will also typecheck if the x argument usage is replaced with ω (instead of 1).

2.6.2 Church-Encoded Natural Numbers

The dependent Church-encoded natural n, where the successor function s is applied n times, can be typed as (Fig. 5):

$$\vdash \lambda t.\lambda s.z.s...sz : (s : ((a : t) \to t)) \to (z : t) \to t.$$

Fig. 5. Church-encode n

This will also typecheck if the s argument usage is replaced with ω (instead of n for some specific n).

3 Towards Compilation to Michelson

The erased core language can be compiled to Michelson by fairly standard procedure, with accommodations for the particular cost model of Michelson—the main addition is the more efficient stack manipulation enabled by usage accounting.

3.1 Stack Tracking

As is standard for compilation of the lambda calculus to stack machines, we track a virtual symbolic stack which maps variable names to stack positions. When a function call is compiled, such as:

```
let f x y = x * y
```

x and y are fetched from their positions in the stack and the body of f is inlined (suppose x is at stack position 3 and y is at stack position 4:

```
{DUG 3; DUP; DIG 4; DUG 5; DUP; DIG 6; MUL}
```

Lambdas in Michelson are quite expensive—each can take only one argument, so multiple-argument functions compiled to lambdas must tuple their arguments before calling the function, and the function body must un-tuple them—so we inline aggressively and also track virtual closures on the stack to avoid compiling to `LAMBDA` whenever possible. All of this is standard fare.

3.2 Usage Accounting

Consider the following indicative example—compilation of the identity function:

```
let f x = x
```

In a normal compilation of the lambda calculus to a stack machine without quantitative type theory or any notion of linearity, x must be kept on the stack in case it is used elsewhere and only dropped after the computation is complete, so f must be compiled to:

```
{DUG 5; DUP; DIG 6}
```

With quantitative type theory, the compiler can lookup the usage annotation for x, and if x is only used once, then x can simply be moved from lower in the stack instead:

```
{DUG 5}
```

This technique easily generalises to multi-argument functions and any usage on the semiring—in cases of usage ω, the non-quantitative behaviour is preserved, and x is instead dropped after the computation is complete.

3.3 Usage Propagation

Consider the following function which uses its argument twice:

```
let f x = x * x
```

Suppose that x is five slots down in the stack, with a total usage of 3. A naive implementation without lookup caching might fetch x twice:

```
{DUG 5; DUP; DIG 6; DUG 6; DUP; DIG 7; MUL}
```

Or, alternatively, with lookup caching but without linearity, x might be duplicated more than necessary (as each lookup must treat the variable as possibly being used elsewhere):

```
{DUG 5; DUP; DIG 6; DUP; DUP; DIG 2; MUL; DIP {DROP}}
```

Instead, with usage annotations, we can propagate two usages of x upwards immediately and avoid both the double-fetch and the unnecessary duplication/cleanup:

```
{DUG 5; DUP; DIG 6; DUP; MUL}
```

4 Future Work

4.1 Improved Usage Accounting with ANF

As detailed in a blog post [14], we plan to add an administrative normal form transformation, such that all functions take primitives—for example, ANF would transform

```
f a (x + y) b
```

into

```
let xy0 = x + y in f a xy0 b
```

This would allow all usages of variables to be moved forward to the top of the stack when used and remaining uses to be moved back, instead of moving all usages but one forward, which is currently required.

4.2 First-Class Usages

Work is in progress to add dependent usages [15], where terms can be lifted into usages and usages can be converted to terms, such that usages can depend on terms in the usual dependent-type-theory sense. This will allow more precise usage accounting in cases where an annotation of ω would otherwise be required, such as where the usage of one argument to a function depends on the value of another argument, although it requires more complex accounting in the compiler.

Acknowledgements. This paper describes part of the ongoing research work being undertaken to develop the Juvix smart contract language [16,17] by the Juvix team at Metastate, including Marty Stumpf, Jeremy Ornelas, Andy Morris, and April Goncalves. Thanks to an anonymous reviewer for comments and suggestions.

References

1. S. Developers, Solidity: An object-oriented, high-level language for implementing smart contracts. https://solidity.readthedocs.io/en/v0.6.8/
2. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf
3. Parity Technologies: A postmortem on the parity multi-sig library self-destruct. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/
4. 0x Core Team, "Post-mortem: 0x v2.0 exchange vulnerability." https://blog.0xproject.com/post-mortem-0x-v2-0-exchange-vulnerability-763015399578
5. Goodman, L.M.: Tezos - a self-amending crypto-ledger, September 2014. https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf
6. Bernardo, B., Cauderlier, R., Pesin, B., Tesson, J.: Albert, an intermediate smart-contract language for the tezos blockchain (2020). https://arxiv.org/abs/2001.02630
7. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-cho-coq, a framework for certifying tezos smart contracts. https://arxiv.org/abs/1909.08671 (2019)
8. Pettersson, J.: Safer smart contracts through type-driven development. https://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf
9. Brady, E.: IDRIS - systems programming meets full dependent types. In: PLPV 2011 - Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification, pp. 43–54 (2011)
10. Edgington, B.: Ethereum lisp like language. https://lll-docs.readthedocs.io/en/latest/lll_introduction.html
11. Atkey, R.: Syntax and semantics of quantitative type theory. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 56–65 (2018)
12. N. Labs: Michelson: The language of smart contracts in tezos. https://tezos.gitlab.io/whitedoc/michelson.html
13. McBride, C.: I got plenty o' Nuttin'. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 207–233. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_12
14. Ornelas, J.: Compiling Juvix to Michelson, May 2020. https://research.metastate.dev/juvix-compiling-juvix-to-michelson/
15. Goes, C., Morris, A.: Usage polymorphism and dependent usages in Juvix, September 2019. https://github.com/cryptiumlabs/juvix/issues/87
16. Goes, C.: The why of Juvix: on the design of smart contract languages, January 2020. https://research.metastate.dev/the-why-of-juvix-part-1-on-the-design-of-smart-contract-languages/
17. Goes, C.: The why of Juvix: Ingredients & architecture, January 2020. https://research.metastate.dev/the-why-of-juvix-ingredients-architecture/