From Call-by-push-value to Stack-Based TAL?

Max S. New
Northeastern University

***Abstract*—We draw an analogy between Stack-based typed assembly language and Call-by-push-value, in particular that STAL stack types are the dual view on CBPV's computation types. This leads to a factorization of much of the translation of call-by-value languages into STAL, in particular the CPS-like interpretation of function types can be expressed in CBPV as the combination of the call-by-value translation and the Church-encoding of the F type using the U type. The gaps in the analogy lead to new questions for both sides. Finally, we report on work-in-progress on designing an assembly language directly based on CBPV structure.**

I. Computation Types and Stack Types

Call-by-push-value (CBPV) [4] is a calculus introduced by Levy as a "subsuming paradigm" for effectful computation, supporting fully complete embeddings of Call-by-value and Call-by-name λ calculus. Semantically, CBPV is based on the decomposition of a strong monad into a strong adjunction, generalizing Moggi's monadic semantics of effects [5]. CBPV has a simple equational theory that make it sound and complete for this categorical semantics. The key to its generality is the introduction of a distinction between *value* types and *computation* types.

Stack-based Typed Assembly Language (STAL) [6] was introduced to give a typed operational view of a clean but still realistic assembly language, extending the earlier TAL to account for stack-based allocation and calling conventions. To do this, two kinds of types are introduced: ordinary types which classify word-sized data, and stack types which classify the structure of the stack.

While approaching the problem from two different perspectives on typed languages, STAL and CBPV introduce essentially the *same* distinction, but with dual presentation: STAL's stack types can be seen as classifying the stack that a corresponding CBPV computation runs against [1]. In particular, the STAL stack constructor $\tau :: \sigma$, the type of stacks consisting of a word classified by τ followed by a stack classified by σ corresponds to the CBPV computation type $A_\tau \to B_\sigma$ that classifies computations that pop an A_τ off the stack and proceed as B_σ. Clearly a precondition on such a computation is that the stack consist of an A_τ and a stack for B_σ. Next, ignoring registers besides the stack-pointer, STAL's *code pointer* type $\forall[\Delta].\{\mathtt{sp} : \sigma\}$ corresponds to a combination of polymorphism with CBPV's *thunk* type: $U(\forall\Delta.B_\sigma)$[2].

Notably missing from STAL is a connective corresponding to CBPV's FA type, which classifies computations that perform effects and (possibly) return A values. However, in the presence of parametric polymorphism, it is possible to Church-encode the F type using the U type [7]:

$$FA \cong \forall Y.U(A \to Y) \to Y$$

Then we see that the elaboration of the CBV function type, when combined with this Church-encoding and a harmless move of a quantifier:

$$A \to_{cbv} A' = U(A \to FA') \cong U(\forall Y.A \to U(A' \to Y) \to Y)$$

corresponds to a simple stack-based calling convention expressible in STAL:

$$\forall[\rho]\{\mathtt{sp} : \sigma :: \forall[]\{\mathtt{sp} : \sigma' :: \rho\} :: \rho\}$$

Notably, both cases easily extend to multi-argument functions.

While these similarities are promising, both languages include connectives missing from the other. On the one hand, CBPV also includes a computation product which we write as $B\&B'$ that acts like a lazy pair. The corresponding stack type would be a *tagged sum of stacks* $\sigma \oplus \sigma'$, a stack consisting of a tag that indicates which case it is and a stack with type corresponding to that tag. Furthermore, CBPV can naturally be extended *recursive* computation types, which can be used to model calling conventions where stack frames have unbounded size, such as variable-arity functions.

On the other hand, STAL includes *compound stacks* $\sigma@\sigma'$ that classifies stacks which consist of a σ stack followed by a σ' stack. This suggests making CBPV computation types *monoidal*, which would then make them equivalent to the linear types in Linear-non-linear Logic [3], however the use of compound stacks in STAL is somewhat restricted, and so we do not consider them as compelling as other features. A bigger difference is that STAL includes *registers*, which are suggestive of a kind of function type in CBPV with *named arguments*.

II. Towards a CBPV TAL

CBPV and STAL both have their advantages. STAL is low-level enough that code generation is nearly trivial, and its support for registers means it can describe a wide variety of calling conventions. CBPV on the other hand has a simple, canonical equational theory with full completeness results for CBV and CBN. This naturally leads us to the question of how to make a more low-level version of CBPV, but maintaining the nice semantic properties of the original CBPV. Such a language could serve as a common low level target language supporting fully abstract compilation from a variety of source languages, as envisioned in [2].

[1] this duality is explored further in work on the Enriched Effect Calculus [8]

[2] technically the U type corresponds to a closure in STAL, but correctly handling this does have to change any of the type translations we show in this abstract

$$
\begin{array}{rcl}
A & ::= & i64 \mid A \times A \mid UB \mid \cdots \\
B & ::= & A \to B \mid \forall Y.B \mid \cdots \\
i & ::= & \texttt{add} \mid \texttt{mult} \mid \texttt{cons} \mid \texttt{uncons}\ \Gamma \mid B \vdash i : B' \\
 & & \mid \texttt{push}x \mid \texttt{push}n \mid \texttt{push}T \\
M & ::= & i; M \mid \lambda x.M \mid \Lambda X.M \mid \texttt{jmp}
\end{array}
$$

$$\Gamma \vdash M : B$$

Fig. 1. Linearized CBPV

In Figure 1, we present a fragment of a version of CBPV that is lower level in that it has a notion of *instruction* and construction of compound values has to be compiled to a sequence of instruction statements. CBPV computations are typed as usual as $\Gamma \vdash M : B$, but we also include a form for instructions, which are typed as $\Gamma \mid B \vdash i : B'$, and can be prepended to a computation with a semi-colon:

$$\frac{\Gamma \vdash M : B \qquad \Gamma \mid B \vdash i : B'}{\Gamma \vdash i; M : B'}$$

The judgmental structure for instructions is the same as that of *stack terms* in CBPV and this sequential composition corresponds to the admissible rule of stack "piling" in CBPV [4].

In CBPV the only way compound values appear in computations are in function application and return statements. We Church-encode the F type and so do not have a primitive return statement, and so only need to compile the function application form, which is the titular *push value* form. In our linearized CBPV, we restrict the push value form to only push variables and 64-bit integer literals. All other manipulation of values is performed through adding *instructions*. For instance, to add value pair introduction and elimination we can simply add instructions that construct and deconstruct tuples at the top of the stack:

$$(A_1 \times A_2) \to B \vdash \texttt{cons} : A_1 \to A_2 \to B$$

$$A_1 \to A_2 \to B \vdash \texttt{uncons} : (A_1 \times A_2) \to B$$

We can similarly add operations of addition and multiplication of integer values.

All computations end in a jump, which we make stack-oriented as well by requiring the destination to be the first operand on the stack:

$$\Gamma \vdash \texttt{jmp} : UB \to B$$

Which corresponds precisely to the co-unit of the adjunction between F and U.

Then we can define a simple operational semantics on abstract machine states $\langle M \mid \gamma \mid S \rangle$ where $\Gamma \vdash M : B$ is the current computation, $\gamma : \Gamma$ is the current environment and $B \vdash S$ is the state of the stack.

Compilation from CBPV to the linearized form is simple, but includes arbitrary choices of evaluation order for construction of compound values. We are working on an equational theory that would prove that any order is equivalent, which we can design based on the obvious *back-translation* from linearized form to ordinary CBPV. There should also be a simulation theorem between the operational semantics of the two languages.

Our next steps are to consider extensions of both linearized and ordinary CBPV with registers. We expect that the correspondence between stack forms and instructions will still hold in this extension. We are also interested in considering *dependently typed* extensions of these languages, based on recent work [1], [9], and making a formal connection to dependently typed TAL [10].

REFERENCES

[1] Daniel Ahman, Neil Ghani, and Gordon D. Plotkin. *Dependent Types and Fibred Computational Effects*. 2016.

[2] Amal Ahmed. Verified Compilers for a Multi-Language World. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15–31, 2015.

[3] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *Selected Papers from the 8th International Workshop on Computer Science Logic*, pages 121–135, 1995.

[4] Paul Blain Levy. *Call-by-Push-Value*. Ph. D. dissertation, Queen Mary, University of London, London, UK, March 2001.

[5] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[6] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.

[7] Rasmus Ejlers Mgelberg and Alex Simpson. Relational Parametricity for Computational Effects. *Logical Methods in Computer Science*, Volume 5, Issue 3, 2009.

[8] Rasmus Ejlers Mgelberg and Sam Staton. Linear usage of state. *Logical Methods in Computer Science*, Volume 10, Issue 1, 2014.

[9] Matthijs Vákár. In search of effectful dependent types. *CoRR*, 2017.

[10] Hongwei Xi and Robert Harper. Dependently typed assembly language. In *ICFP*, pages 169–180, September 2001.