Towards Behavioral Compiler Correctness Proofs using Hidden Logic

Lutz Hamel[1] and Grigore Roşu[2]

[1] Department of Computer Science and Statistics
University of Rhode Island, Kingston, R.I. 02881, USA
email: hamel@cs.uri.edu
[2] Research Institute for Advanced Computer Science
NASA Ames Research Center, Moffett Field, CA 94035, USA
email: grosu@ptolemy.arc.nasa.gov

Abstract. A user will perceive a compiler as correct if he or she cannot visibly distinguish between the behavior of a program interpreted in the source language semantics and the behavior of the compiled version of the same program with the exception of the execution speed. This allows us to use the visible behavior of a program as a correctness argument for program translation. Recent developments in behavioral specification and proving enable us to reason about visible behavior effectively and automatically. We introduce a novel notion of *behavioral refinement* and develop the concepts of *behavioral language specification* and *language morphism*, together with their model theoretic interpretation.

1 Introduction

We write programs to effect observable behavior on computing machinery, such as printing the results of a computation on a terminal screen or writing processed data to a file. To keep programs extendable and maintainable, their development is typically done in a high level programming language. Execution of programs by interpreting the high level syntactic structures directly will achieve the desired visible behavior, but usually interpretation is too slow for most applications. It is therefore necessary to map the high level language structures into a lower level, more efficient setting, such as a virtual machine or assembly language. In general, the user does not care how this mapping or compilation is accomplished as long as it preserves the intended visible behavior and reduces the overall execution time. This line of reasoning allows us to use visible program behavior as a correctness argument for compilers, that is, *a compiler is correct iff it preserves the visible behavior of the source language.*

Here we formalize this notion of compiler correctness by introducing equational behavioral language specifications with behavior preserving language morphisms between them. The syntactic part of a language morphism can be considered a behavior preserving compiler specification. All traditional programming language features can be described with equational specifications [3, 13, 30].

These specification techniques extend seamlessly to behavioral language specifications as well [17].

We develop our theoretical foundations in the context of hidden logic. Hidden logics refer to approaches to hidden algebra where the main intuitions of splitting sorts into visible and hidden sorts as well as behavioral satisfaction of equations is preserved and the models are still considered to be hidden algebras, but other notions such as behavioral specification and refinement are modified to fit the requirements of the particular application [24]. Typically these logics do not yield an institution [10] but are appropriate for the problems at hand. This is also the case here, where we develop a hidden logic with its corresponding notions of behavioral specification and refinement appropriate for our compiler correctness problem.

Recent advances in automated behavioral reasoning such as *circular coinductive rewriting* allow us to mechanize behavioral proofs. In this paper we use the BOBJ behavioral specification system for our behavioral language specifications and behavioral proofs. BOBJ implements behavioral rewriting and circular coinductive rewriting [12, 24].

We used an earlier version of the framework developed here for the behavioral correctness proof of a fairly efficient compiler for a substantial subset of the OBJ3 algebraic specification language [16, 17]. In this paper we take a look at a compiler that compiles a simple assignment language into accumulator based machine code.

The paper is structured as follows. Section 2 defines some technical preliminaries. In Section 3 we develop our core algebraic and behavioral notions. Behavioral language specifications and language morphisms are introduced in Section 4. Section 5 discusses the behavioral correctness proof of the assignment language compiler. Section 6 discusses related work. Finally, we conclude with Section 7.

2 Preliminaries

Due to space limitations, we assume the reader is familiar with general notions of algebra and equational logics, such as initial and reduct algebras, morphisms, and satisfaction. Familiarity with category theory helps but is not crucial.

To summarize briefly: If $\varphi\colon\ \Sigma \to \Sigma'$ is a signature morphism and A' is a Σ'-algebra, then we let $A'|_\varphi$ denote the φ-reduct of A' to a Σ-algebra. $\mathbf{Alg}_\Sigma$ denotes the category of Σ-algebras and Σ-morphisms; $\_|_\varphi$ is then a functor $\mathbf{Alg}_{\Sigma'} \to \mathbf{Alg}_\Sigma$. If e is a Σ-equation then $\varphi(e)$ is its translation to a Σ'-equation. It is known under the name "satisfaction condition" [10] that in the context above, $A' \models_{\Sigma'} \varphi(e)$ iff $A'|_\varphi \models_\Sigma e$. If t is a Σ-term and A is a Σ-algebra, then $A_t\colon\ A^{var(t)} \to A$ is the interpretation of t in A: for any map $\theta\colon\ var(t) \to A$, $A_t(\theta)$ is the evaluation of t in A where all the variables in t are replaced by their concrete values given by θ. If a variable of t, say $\star$, is of special importance, then we can view the evaluation of t in two steps, that is, $A_t\colon\ A \to (A^{(var(t)-\{\star\})} \to A)$ with the obvious meaning. Given any many-sorted signature Σ, a *derived*

operation $\delta : s_1...s_n \to s$ of Σ is a term in $T_{\Sigma,s}(\{z_1, ..., z_n\})$, where $z_1, ..., z_n$ are special variables[1] of sorts $s_1, ..., s_n$. For any Σ-algebra A, the interpretation of δ in A is the map $A_\delta\colon A_{s_1} \times \cdots \times A_{s_n} \to A_s$ defined as $A_\delta(a_1, ..., a_n) = \theta(\delta)$, where $\theta\colon \{z_1, ..., z_n\} \to A$ takes z_i to a_i for all $i = 1...n$. We let $Der(\Sigma)$ denote the signature of all derived operations of Σ.

3 Hidden Logic

Hidden algebra extends algebraic specification to handle states naturally, using behavioral equivalence and satisfaction. Systems need only satisfy their requirements behaviorally, in the sense of *appearing* to satisfy them under all possible observable experiments that can be performed. One distinctive feature of hidden algebra is to split sorts into *visible* sorts for data and *hidden* sorts for states. A model, or *hidden algebra*, is an abstract implementation, consisting of the possible states, with functions for attributes and methods (attributes "observe" states by returning visible values, while methods "modify" states). Intuitively, the hidden objects are black-boxes from which one can extract information via *experiments*. Hidden algebra was introduced in [9] to give algebraic semantics to the object paradigm, and advanced further in [11, 14, 25, 15] among other places.

Hidden (or behavioral) logics refer to close relatives of hidden algebra, including also *observational logic* [1, 2, 19] and *coherent hidden algebra* [7, 8]. A detailed presentation of various hidden logics appears in [24], together with relations to many other concepts, history, and proofs for results mentioned but not proved here. Two important classes of hidden logics can be distinguished, called *fixed-data* and *loose-data*, respectively, depending on whether the data universe is assumed fixed or not.

Here we formally define a loose-data hidden logic that we find appropriate to specify and prove the behavioral correctness of compilers. While the main intuitions of hidden algebra remain the same, i.e., sorts are split into visible/hidden and *behavioral equivalence* is defined as "indistinguishability under experiments," we consider *three* types of operations in a hidden signature: 1) operations that generate the experiments; 2) operations that are compatible with the generated behavioral equivalence, and also 3) operations which do not preserve the behavioral equivalence. An example of the third is an attribute that observes the execution environment of a program: two programs can be behaviorally equivalent even if the variable assignments are different when they terminate. We also had to extend the notion of hidden theory morphisms, since earlier definitions [15, 24] seemed to be too restrictive for our approach to compiler correctness. The following formalizes these notions.

3.1 Hidden Signature, Contexts and Experiments

Hidden signatures split the sorts into visible and hidden, the visible sorts denoting data while the hidden ones denote states.

[1] These are assumed different from any other variables in a given situation.

Definition 1. *Given disjoint sets V, H called* **visible** *and* **hidden sorts**, *a* **hidden (V,H)-signature** *is a many sorted $(V \cup H)$-signature. A* **hidden subsignature of** Σ *is a hidden (V,H)-signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma\restriction_V = \Sigma\restriction_V$.* $\mathbf{HAlg}_\Sigma$ *is the category of Σ-algebras, called* **hidden Σ-algebras**.

We next formalize the notion of "experiment," which informally is an observation of an attribute of a system after it has been perturbed. Only a reduced number of operations are used in experiments. The symbol $\star$ is a placeholder for the state being experimented upon.

Definition 2. *Given $\Delta \subseteq Der(\Sigma)$, a* **$\Delta$-context for sort** $h \in H$ *is a term in $T_\Delta(\{\star : h\} \cup Z)$, where Z is an infinite set of special variables. Let $\mathcal{C}_\Delta[\star : h]$ denote the set of all Δ-contexts for sort h. A Δ-context with visible result sort is called a* **Δ-experiment**; *$\mathcal{E}_\Delta[\star : h] \subseteq \mathcal{C}_\Delta[\star : h]$ is the set of all Δ-experiments for sort h. If $c \in \mathcal{C}_\Delta[\star : h]$ and $t \in T_{\Sigma,h}(X)$, then $var(c)$ denotes the variables of c but $\star$, and $c[t]$ denotes the term obtained from c by substituting t for $\star$.*

It is frequently the case in the literature that contexts have *exactly one* occurrence of $\star : h$; however, this requirement is not needed here.

3.2 Behavioral Equivalence

We next define a distinctive feature of hidden logic. Two states are behaviorally equivalent iff they are indistinguishable by experiments:

Definition 3. *Given a Σ-algebra A and a Δ as above, the equivalence which is identity on visible sorts and $a \equiv^\Delta_\Sigma a'$ iff $A_\gamma(a)(\theta) = A_\gamma(a')(\theta)$ for all Δ-experiments γ and all maps $\theta\colon var(\gamma) \to A$ is called* **Δ-behavioral equivalence on** *A. Given an equivalence $\sim$ on A, an operation $\sigma : s_1 ... s_n \to s$ is* **congruent for** *$\sim$ iff $A_\sigma(a_1, ..., a_n) \sim_s A_\sigma(a'_1, ..., a'_n)$ whenever $a_i \sim_{s_i} a'_i$ for $i = 1...n$. σ is* **Δ-behaviorally congruent for** *A iff it is congruent for $\equiv^\Delta_\Sigma$. A* **hidden Δ-congruence on** *A is an equivalence on A which is the identity on visible sorts and for which each derived operation in Δ is behaviorally congruent.*

We may write $\equiv$ instead of $\equiv^\Delta_\Sigma$ when Σ and Δ can be inferred from context, write $\equiv_\Sigma$ when $\Sigma = \Delta$, and "congruent" instead of "behaviorally congruent." It can be shown that $\equiv^\Delta_\Sigma$ is the largest hidden Δ-congruence [25, 24].

It can be shown that behavioral equivalence is the largest hidden Δ-congruence [25, 24], and that this does *not* depend upon the existence of final models (which may not even exist [5, 24]) as in coalgebra [20] and not even upon a fixed data universe as in original hidden algebra [14]. This is the basis for *coinduction*, which we do not discuss here.

3.3 Behavioral Satisfaction

Unlike satisfaction in typical equational formalisms, hidden logic only requires that its models (hidden algebras) satisfy the requirements behaviorally, in the sense of "appearing" to be satisfied under any possible experiment:

Definition 4. *A* Δ**-behaviorally satisfies** *a* Σ*-equation* $(\forall X)\ t = t'$*, say* e*, iff* $\theta(t) \equiv^{\Delta}_{\Sigma} \theta(t')$ *for each* $\theta\colon X \to A$*; we write it* $A \models^{\Delta}_{\Sigma} e$*. If* E *is a set of* Σ*-equations,* $A \models^{\Delta}_{\Sigma} E$ *iff* A Δ*-behaviorally satisfies each* Σ*-equation in* E*.*

When Σ and Δ are clear, we may write $\models$ instead of $\models^{\Delta}_{\Sigma}$. The theory also allows conditional equations [14, 15, 24], but they are not needed here.

Definition 5. *Given a* Σ*-equation* $(\forall X)\ t = t'$*, say* e*, let* $\mathcal{E}_\Delta[e]$ *denote either the set of* Σ*-equations* $\{(\forall X, var(\gamma))\ \gamma[t] = \gamma[t'] \mid \gamma \in \mathcal{E}_\Delta[\star : h]\}$ *when the sort* h *of* t *and* t' *is hidden, or the set* $\{e\}$ *when the sort of* t *and* t' *is visible. If* E *is a set of* Σ*-equations then let* $\mathcal{E}_\Delta[E]$ *be the set* $\bigcup_{e \in E} \mathcal{E}_\Delta[e]$.

The following result in [24] says that behavioral satisfaction of an equation can be reduced to strict satisfaction of a potentially infinite set of equations:

Proposition 1. *If* A *is a hidden* Σ*-algebra then* $A \models^{\Delta}_{\Sigma} E$ *iff* $A \models_\Sigma \mathcal{E}_\Delta[E]$.

3.4 Behavioral Specification

The following definition of behavioral specification is novel and appears as a consequence of our efforts to behaviorally specify and verify compilers. It generalizes previous formulations of behavioral specification [11, 14, 25, 24]:

Definition 6. *A* **behavioral** *(or* **hidden***)* Σ**-specification** *(or* **-theory***) is a tuple* $(\Delta, \Sigma, \Gamma, E)$ *where* Σ *is a hidden signature and* Δ *as above,* Γ *is a hidden subsignature of* Σ*, and* E *is a set of* Σ*-equations. The operations in* $\Gamma - \Sigma\restriction_V$ *are called* **behavioral***. We usually let* $\mathcal{B}$*,* $\mathcal{B}'$*,* $\mathcal{B}_1$*, etc., denote behavioral specifications. A hidden* Σ*-algebra* A **behaviorally satisfies** *(or* **is a model of***)* *a behavioral specification* $\mathcal{B} = (\Delta, \Sigma, \Gamma, E)$ *iff* $A \models^{\Delta}_{\Sigma} E$ *and all the operations in* Γ *are behaviorally congruent, and in this case we write* $A \models \mathcal{B}$*; we write* $\mathcal{B} \models e$ *if* $A \models \mathcal{B}$ *implies* $A \models^{\Delta}_{\Sigma} e$ *for all* A*. An operation* $\sigma \in \Sigma$ *is* **behaviorally congruent for** $\mathcal{B}$ *if and only if it is congruent for any* $A \models \mathcal{B}$*. The full subcategory of* $\mathbf{HAlg}_\Sigma$ *of hidden algebras satisfying* $\mathcal{B}$ *is denoted by* $\mathbf{HAlg}_\mathcal{B}$.

In [11, 14] and many other places it was assumed that $\Delta = \Gamma = \Sigma$, while in [25, 24] it is assumed that only $\Delta = \Gamma$ and congruence criteria are given allowing the automatic calculation of a small Δ from Γ. The operations in Γ are called "behavioral" because they preserve the behavioral equivalence.

3.5 Behavioral Refinement

The following concept of behavioral refinement is also new. Other notions of morphisms were given in [15, 24], but they are too strong for our purpose.

Definition 7. *Given two behavioral specifications* $\mathcal{B} = (\Delta, \Sigma, \Gamma, E)$ *and* $\mathcal{B}' = (\Delta', \Sigma', \Gamma', E')$ *over the same data algebra, a signature morphism* $\varphi\colon \Sigma \to Der(\Sigma')$ *is called a* **behavioral refinement** *if and only if*

1. φ *is the identity on* $\Sigma\restriction_V$*,*
2. $\varphi(\Delta) \subseteq Der(\Delta')$*,*

3. *the operations in* $\varphi(\Gamma)$ *are* Δ'*-behaviorally congruent,*
4. *for each hidden sort* $h \in H$ *and each* $\gamma' \in \mathcal{E}_{\Delta'}[\star : \varphi(h)]$ *there is some* $\gamma \in \mathcal{E}_{\Delta}[\star : h]$ *such that* $\mathcal{B}' \models (\forall var(\gamma'), \star : \varphi(h))\ \gamma' = \varphi(\gamma)$*, and*
5. $\mathcal{B}' \models \varphi(E)$.

We let $\varphi : \mathcal{B} \to \mathcal{B}'$ *denote a behavioral refinement as defined above.*

Notice that 4. above follows automatically when $\varphi\restriction_{\Delta} : \Delta \to Der(\Delta')$ is surjective on Δ', as it is the case in [15, 24] and usually in practice.

Proposition 2. *If* $\varphi : \mathcal{B} \to \mathcal{B}'$ *is a behavioral refinement,* A' *is a hidden* Σ'*-algebra with* $A' \models \mathcal{B}'$*, and* $\equiv^{\Delta}_{\Sigma}$ *is the* Δ*-behavioral equivalence on* $A'\restriction_{\varphi}$*,*

1. $\equiv^{\Delta}_{\Sigma,s} = \equiv^{\Delta'}_{\Sigma',\varphi(s)}$ *for any sort* s *in* Σ*,*
2. $\sigma \in \Sigma$ *is congruent for* $\equiv^{\Delta}_{\Sigma,s}$ *iff* $\varphi(\sigma)$ *is congruent for* $\equiv^{\Delta'}_{\Sigma',\varphi(s)}$*,*
3. $A' \models^{\Delta'}_{\Sigma'} \varphi(e)$ *iff* $A'\restriction_{\varphi} \models^{\Delta}_{\Sigma} e$ *for any* Σ*-equation* e*, and*
4. $A'\restriction_{\varphi} \models \mathcal{B}$.

Proof. 1. Notice that it holds when s is visible because both relations are identity relations on $A'_{\varphi(s)}$. Assume then that s is hidden. If $a, a' \in A'_{\varphi(s)}$ such that $a \equiv^{\Delta'}_{\Sigma',\varphi(s)} a'$, then for any experiment $\gamma \in \mathcal{E}_{\Delta}[\star : s]$ it is the case that $(A'\restriction_{\varphi})_{\gamma}(a) = (A'\restriction_{\varphi})_{\gamma}(a')$ because all the operations in $\varphi(\Gamma)$ are Δ'-behaviorally congruent, so $A'_{\varphi(\gamma)}(a) = A'_{\varphi(\gamma)}(a')$; hence, $a \equiv^{\Delta}_{\Sigma,s} a'$. Conversely, if $a \equiv^{\Delta}_{\Sigma,s} a'$ then let γ' be any Δ'-experiment in $\mathcal{E}_{\Delta'}[\star : \varphi(s)]$ and let γ be a Δ-experiment in $\mathcal{E}_{\Delta}[\star : s]$ such that $\mathcal{B}' \models (\forall var(\gamma'), \star : \varphi(s))\ \gamma' = \varphi(\gamma)$. Then $A'_{\gamma'}(a) = A'_{\varphi(\gamma)}(a) = (A'\restriction_{\varphi})_{\gamma}(a)$ and similarly $A'_{\gamma'}(a') = (A'\restriction_{\varphi})_{\gamma}(a')$. It follows now immediately that $A'_{\gamma'}(a) = A'_{\gamma'}(a')$, so $a \equiv^{\Delta'}_{\Sigma',\varphi(s)} a'$.

2. It follows from *1.*, noticing that $(A'\restriction_{\varphi})_{\sigma} = A'_{\varphi(\sigma)}$.

3. If the sort of e is visible then the result is straightforward. Suppose then that e has a hidden sort h. If $A' \models^{\Delta'}_{\Sigma'} \varphi(e)$ then by Proposition 1 one obtains $A' \models_{\Sigma'} \mathcal{E}_{\Delta'}[\varphi(e)]$. But $\varphi(\mathcal{E}_{\Delta}(e)) \subseteq \mathcal{E}_{\Delta'}(\varphi(e))$ because $\varphi(\Delta) \subseteq Der(\Delta')$, so $A' \models_{\Sigma'} \varphi(\mathcal{E}_{\Delta}(e))$. It follows now by the satisfaction lemma of equational logic that $A'\restriction_{\varphi} \models_{\Sigma} \mathcal{E}_{\Delta}(e)$, and again by Proposition 1 that $A'\restriction_{\varphi} \models^{\Delta}_{\Sigma} e$. Conversely, suppose that $A'\restriction_{\varphi} \models^{\Delta}_{\Sigma} e$. Then by Proposition 1 it follows that $A'\restriction_{\varphi} \models_{\Sigma} \mathcal{E}_{\Delta}(e)$. Let $\gamma' \in \mathcal{E}_{\Delta'}[\star : \varphi(h)]$ and let $\gamma \in \mathcal{E}_{\Delta}[\star : h]$ such that $\mathcal{B}' \models (\forall var(\gamma'), \star : \varphi(h))\ \gamma' = \varphi(\gamma)$. Since $A' \models \mathcal{B}'$, it follows that $A'_{\gamma'} = A'_{\varphi(\gamma)}$, so $A' \models_{\Sigma'} \gamma'[\varphi(e)]$ iff $A' \models_{\Sigma'} (\varphi(\gamma))[\varphi(e)]$ iff $A' \models_{\Sigma'} \varphi(\gamma[e])$ iff $A'\restriction_{\varphi} \models_{\Sigma} \gamma[e]$. Therefore, $A' \models_{\Sigma'} \mathcal{E}_{\Delta'}[\varphi(e)]$. By Proposition 1 it follows now that $A' \models^{\Delta'}_{\Sigma'} \varphi(e)$

It is interesting to note that this result does not depend on $\varphi\restriction_{\Delta} : \Delta \to Der(\Delta')$ (from Definition 7) to be surjective on Δ'.

4. The fact that all the operations in Γ are behaviorally congruent for $A'\restriction_{\varphi}$ is immediate from *2.* and the fact that $A'\restriction_{\varphi} \models^{\Delta}_{\Sigma} E$ follows from *3.*

Thus, a behavioral refinement $\varphi : \mathcal{B} \to \mathcal{B}'$ yields a reduct functor $\_\restriction_{\varphi} : \mathbf{HAlg}_{\mathcal{B}'} \to \mathbf{HAlg}_{\mathcal{B}}$. A natural question is whether behavioral refinements compose:

Proposition 3. *If* $\varphi_1 : \mathcal{B}_1 \to \mathcal{B}_2$ *and* $\varphi_2 : \mathcal{B}_2 \to \mathcal{B}_3$ *are behavioral refinements, where* $\mathcal{B}_1 = (\Delta_1, \Sigma_1, \Gamma_1, E_1)$, $\mathcal{B}_2 = (\Delta_2, \Sigma_2, \Gamma_2, E_2)$, *and* $\mathcal{B}_3 = (\Delta_3, \Sigma_3, \Gamma_3, E_3)$, *then* $\varphi_1;\varphi_2 : \mathcal{B}_1 \to \mathcal{B}_3$ *is also a behavioral refinement. Therefore, behavioral specifications and refinements form a category, called* **BSpec**.

Proof. Notice that $\varphi_1;\varphi_2 : \Sigma_1 \to Der(\Sigma_3)$ is well defined for $\varphi_1 : \Sigma_1 \to Der(\Sigma_2)$ and $\varphi_2 : \Sigma_2 \to Der(\Sigma_3)$ as φ_2 can be first extended to $\varphi_2 : Der(\Sigma_2) \to Der(\Sigma_3)$ and then[2] composed with φ_1. It is easy to see that $\varphi_1;\varphi_2$ thus defined is identity on $\Sigma_1\restriction_V = \Sigma_2\restriction_V = \Sigma_3\restriction_V$ and that $(\varphi_1;\varphi_2)(\Delta_1) \subseteq Der(\Delta_3)$. The operations in $(\varphi_1;\varphi_2)(\Gamma_1)$ are Δ_3-behaviorally congruent since for any hidden Σ_3-algebra A_3, the equivalences on A_3, $A_3\restriction_{\varphi_2}$, $(A_3\restriction_{\varphi_2})\restriction_{\varphi_1}$ are related as in *1.* in Proposition 2.

Let h_1 be a hidden sort in Σ_1, γ_3 be in $\mathcal{E}_{\Delta_3}[\star : (\varphi_1;\varphi_2)(h_1)]$, and $A_3 \models \mathcal{B}_3$. Since φ_2 is a refinement, from *4.* in Definition 7, there is an experiment γ_2 in $\mathcal{E}_{\Delta_2}[\star : \varphi_1(h_1)]$ such that $\mathcal{B}_3 \models (\forall var(\gamma_3), \star : (\varphi_1;\varphi_2)(h_1))\ \gamma_3 = \varphi_2(\gamma_2)$, which yields $(A_3)_{\gamma_3} = (A_3)_{\varphi_2(\gamma_2)} = (A_3\restriction_{\varphi_2})_{\gamma_2}$. Since φ_1 is a behavioral refinement, by *4.* in Definition 7 it follows that there is some experiment γ_1 in $\mathcal{E}_{\Delta_1}[\star : h_1]$ such that $\mathcal{B}_2 \models (\forall var(\gamma_2), \star : \varphi_1(h_1))\ \gamma_2 = \varphi_1(\gamma_1)$, which yields $(A_3\restriction_{\varphi_2})_{\gamma_2} = (A_3\restriction_{\varphi_2})_{\varphi_1(\gamma_1)} = ((A_3\restriction_{\varphi_2})\restriction_{\varphi_1})_{\gamma_1} = (A_3\restriction_{(\varphi_1;\varphi_2)})_{\gamma_1} = (A_3)_{(\varphi_1;\varphi_2)(\gamma_1)}$ because of *3.* in Proposition 2. Therefore, $\mathcal{B}_3 \models (\forall var(\gamma_3), \star : (\varphi_1;\varphi_2)(h_1))\ \gamma_3 = (\varphi_1;\varphi_2)(\gamma_1)$.

We show next that $\mathcal{B}_3 \models (\varphi_1;\varphi_2)(E_1)$. Let $A_3 \models \mathcal{B}_3$. Then by *3.* in Proposition 2, $A_3 \models^{\Delta_3}_{\Sigma_3} \varphi_2(\varphi_1(E_1))$ iff $A_3\restriction_{\varphi_2} \models^{\Delta_2}_{\Sigma_2} \varphi_1(E_1)$. Further, by *4.* and *3.* in Proposition 2, $A_3\restriction_{\varphi_2} \models^{\Delta_2}_{\Sigma_2} \varphi_1(E_1)$ iff $(A_3\restriction_{\varphi_2})\restriction_{\varphi_1} \models^{\Delta_1}_{\Sigma_1} E_1$. The latter is true since $A_3\restriction_{\varphi_2} \models \mathcal{B}_2$ and $(A_3\restriction_{\varphi_2})\restriction_{\varphi_1} \models \mathcal{B}_1$.

As noted before, we do not pose any surjectivity constraints of φ on hidden sorts (Definition 7), as in [15]. Therefore, the hidden logic defined above is not an institution in the sense of [10], but we claim that the above is the appropriate notion of behavioral refinement in the case of languages as in the present paper.

3.6 Behavioral Reasoning and BOBJ

Standard equational reasoning is not sound for behavioral satisfaction due to operations which do not preserve the behavioral equivalence. *Behavioral equational deduction* and *behavioral rewriting* modify standard equational deduction and term rewriting to make them sound [7], respectively. In order to prove interesting behavioral properties *context induction* [18] and *hidden coinduction* [14] are more appropriate proof techniques in hidden logics, but unfortunately they both require intensive human intervention. *Circular coinductive rewriting* (CCRW) [12, 24] is a completely automatic and surprisingly powerful method that allowed us to do most of the proofs that we found in the related literature *automatically*. Briefly, Δ-coinduction proves behavioral equalities directly by applying repeatedly all operations in Δ to the top of two terms until they can be shown to be equal by behavioral rewriting. This procedure can loop forever and for this reason, circular coinduction further detects circular patterns in these expansions.

[2] In a similar spirit to composition in Kleisly categories.

BOBJ (Behavioral OBJ) [12, 24] is an executable behavioral specification language that supports hidden logics and circular coinductive rewriting. BOBJ extends OBJ3 [16] by allowing behavioral theories in addition to (visible) algebraic specifications. Behavioral theories are introduced with the keyword `bth`.

We next show how BOBJ works on an example of infinite streams:

```
bth STREAM is pr NAT .
  sort Stream .
  op head : Stream -> Nat .
  op tail : Stream -> Stream .
  op even : Stream -> Stream .
  op odd  : Stream -> Stream .
  op zip  : Stream Stream -> Stream .
  var N : Nat .  var S S' : Stream .
  eq head(odd(S)) = head(S) .
  eq tail(odd(S)) = even(tail(S)) .
  eq head(even(S)) = head(tail(S)) .
  eq tail(even(S)) = even(tail(tail(S))) .
  eq head(zip(S, S')) = head(S) .
  eq tail(zip(S, S')) = zip(S', tail(S)) .
end
```

As usual, `head` and `tail` give the head and the tail of an infinite stream, `odd` and `even` give the streams of elements on odd and even positions, and `zip` merges two streams. The equations of hidden sort are automatically considered behavioral. By default, all the operations are assumed behavioral; operations that are not behaviorally congruent can be declared using the attribute `ncong`. BOBJ implements algorithms that detect automatically a *minimal* Δ as in Definition 6, which is called a *cobasis*; a user has the option to define custom cobases using the command `cobasis`. In our case of `STREAM`, BOBJ finds the cobasis {`head, tail`}. Once a cobasis is found, circular coinductive rewriting is performed as follows using the `cred` keyword:

```
cred zip(odd(S), even(S)) == S .
cred zip(odd(S), even(odd(S))) == S .
```

As expected, the first reduction returns `true`, while the second returns `false`. These proofs are not trivial and BOBJ can do them fully automatically. A proof of the first by context induction would require four user-defined non-trivial lemmas, each in turn also proved by context induction.

4 Languages and Language Morphisms

In general, a programming language specification consists of a description of its syntax as well as a specification of its semantics related by an appropriate mapping. The mapping interprets the syntactic structures of the programming language in the appropriate semantic constructs of the programming language. In the behavioral setting this mapping is a behavioral refinement. Formally,

Definition 8. *A* **behavioral language specification** *is a behavioral refinement* $\mathcal{L} : \mathcal{B}_{Syn} \to \mathcal{B}_{Sem}$*, where* $\mathcal{B}_{Syn}$ *and* $\mathcal{B}_{Sem}$ *are behavioral theories specifying the syntax and semantics of the language, respectively. The* **behavioral language semantics** *of a language specification* $\mathcal{L} : \mathcal{B}_{Syn} \to \mathcal{B}_{Sem}$ *is the reduct functor* $\_\restriction_{\mathcal{L}} : \mathbf{HAlg}_{\mathcal{B}_{Sem}} \to \mathbf{HAlg}_{\mathcal{B}_{Syn}}$*. When there is no confusion we refer to behavioral language specifications as languages.*

Notice the distinction between the semantic specification of a language $\mathcal{L}$, which in the situation above is $\mathcal{B}_{Sem}$, and the semantics of a language specification,

which is the functor $\_\restriction_{\mathcal{L}}\colon \mathbf{HAlg}_{\mathcal{B}_{Sem}} \to \mathbf{HAlg}_{\mathcal{B}_{Syn}}$. This reduct functor takes any correct "implementation" of $\mathcal{B}_{Sem}$ to an implementation of $\mathcal{B}_{Syn}$, i.e., this reduct functor takes any model of the semantics to a model of the syntax of the programming language. This is exactly what we expect when we write programs, we expect that our syntactic constructs have an appropriate model that allow us to effect some computation. The definition above is correct because of Proposition 2.

Our definition of language specification was inspired by the work in [3, 13, 30]. It is also related to the "classical denotational semantics" [27, 28] by the clear separation of syntax and semantics and the use of explicit semantic operators that take each syntactic construct to its appropriate semantic domain.

A language morphism is a behavior preserving mapping from one language to another; it has both a syntactic and a semantic component. Formally,

Definition 9. *Given languages* $\mathcal{L}\colon \mathcal{B}_{Syn} \to \mathcal{B}_{Sem}$ *and* $\mathcal{L}'\colon \mathcal{B}'_{Syn} \to \mathcal{B}'_{Sem}$, *a* **behavioral language morphism** *is a pair of behavioral refinements* $\phi = (\phi_{Syn}, \phi_{Sem})\colon \mathcal{L} \to \mathcal{L}'$, *where* $\phi_{Syn}\colon \mathcal{B}_{Syn} \to \mathcal{B}'_{Syn}$ *and* $\phi_{Sem}\colon \mathcal{B}_{Sem} \to \mathcal{B}'_{Sem}$ *such that the left diagram below*

$$\begin{array}{ccc} \mathcal{B}_{Syn} & \xrightarrow{\mathcal{L}} & \mathcal{B}_{Sem} \\ {\scriptstyle \phi_{Syn}}\downarrow & & \downarrow{\scriptstyle \phi_{Sem}} \\ \mathcal{B}'_{Syn} & \xrightarrow[\mathcal{L}']{} & \mathcal{B}'_{Sem} \end{array} \qquad \begin{array}{ccc} \mathbf{HAlg}_{\mathcal{B}_{Syn}} & \xleftarrow{\restriction_{\mathcal{L}}} & \mathbf{HAlg}_{\mathcal{B}_{Sem}} \\ {\scriptstyle \restriction_{\phi_{Syn}}}\uparrow & & \uparrow{\scriptstyle \restriction_{\phi_{Sem}}} \\ \mathbf{HAlg}_{\mathcal{B}'_{Syn}} & \xleftarrow[\restriction_{\mathcal{L}'}]{} & \mathbf{HAlg}_{\mathcal{B}'_{Sem}} \end{array}$$

commutes. The right diagram of functors between categories of hidden algebras, the semantic counterpart, automatically commutes. Behavioral language specifications and language morphisms form a category which we call **BLang**.

In most practical situations the syntax specifications $\mathcal{B}_{Syn}$ and $\mathcal{B}'_{Syn}$ contain no equations. Also, typically the languages $\mathcal{L}$ and $\mathcal{L}'$ are inclusions of theories. The syntactic component ϕ_{Syn} of the language morphism above can be considered a *compiler specification*.

The above diagrams have a deeper meaning than it might seem at first sight. In fact, they highlight two key insights in our approach: First, given any implementation of the target language $\mathcal{L}'$, say A', then a language morphism ϕ yields an implementation of the source language $\mathcal{L}$, namely $A'\restriction_{\phi_{Sem}}$. That is, an implementation of a target language of a language morphism yields an implementation of the source language of a language morphism. Second, given a program p in the source language $\mathcal{L}$, then p and its compiled version $\phi_{Syn}(p)$ are behaviorally equivalent with respect to $\mathcal{L}$ in any possible implementation of $\mathcal{L}'$, that is, they *appear* to be equal under any possible experiment that can be performed in $\mathcal{L}$. We find these results very natural in that they reflect *exactly* the intuitions behind our concept of compiler correctness stated in the introduction.

One surprising consequence of using behavioral theories for the specification of languages is that *syntax is hidden*! Intuitively, we can view languages as programmable abstract machines where the visible behavior of the machines is due to hidden programs.

5 A Simple Example

An earlier version of this framework based on hidden algebra and context induction as the primary proof methodology was used to behaviorally verify a fairly efficient compiler for the algebraic specification language OBJ3 [17]. Here we demonstrate our current incarnation of our framework by working through a simple example. As the source language we chose an assignment language inspired by the UNIX `bc` utility. The target language is a simple accumulator based machine language. What struck us particularly while working through this example is the relative simplicity of the proofs involved due to the coinductive rewriting machinery introduced by BOBJ compared to the context induction in our earlier approach.

A typical sentence in our source language consists of a list of assignment statements followed by a single print statement ("`<expr> !`") allowing the user to view the value of an expression, e.g., `x := SO; y := SSO; (x + y)!`. Constants are given in Peano notation, where `0` $\mapsto 0$; `SO` $\mapsto 1$; `SSO` $\mapsto 2$; *etc.* The semantics for this language essentially define an abstract interpreter with an environment for associating variables with values and a mechanism for evaluating literals.

The target language consists of a number of machine instructions that manipulate the contents of an accumulator and that support both direct and indirect addressing modes. The above computation in the assignment language can be expressed in our machine language as follows: `loada so; stora x; loada sso; adda x;`. Again, constant expressions are given in Peano notation. The semantics for this language model the accumulator, program counter, and machine memory. It is worthwhile noting that the accumulator is modeled with the attribute `acc` mapping the internal value of the accumulator into a visible value, so that a user can observe that part of the hidden machine state. This attribute is also a member of the cobasis required by the coinductive correctness proofs.

The following is a sketch of the behavioral correctness proof of our compiler. Let $\mathcal{P}\colon\ (\Sigma^{\mathcal{P}}_{Syn}, \emptyset) \hookrightarrow (\Sigma^{\mathcal{P}}_{Sem}, E^{\mathcal{P}}_{Sem})$ and $\mathcal{M}\colon\ (\Sigma^{\mathcal{M}}_{Syn}, \emptyset) \hookrightarrow (\Sigma^{\mathcal{M}}_{Sem}, E^{\mathcal{M}}_{Sem})$ be the language specifications for the source programming language and machine language specifications, respectively. Note that in both cases the syntactic theories are empty and the mappings $\mathcal{P}$ and $\mathcal{M}$ are inclusions. Let $\phi^{\mathcal{PM}} = (\phi^{\mathcal{PM}}_{Syn}, \phi^{\mathcal{PM}}_{Sem})\colon\ \mathcal{P} \to \mathcal{M}$ be a mapping from the source programming language to our machine language. We need to show that this mapping constitutes a behavioral language morphism. Formally,

Proposition 4. *$\phi^{\mathcal{PM}} = (\phi^{\mathcal{PM}}_{Syn}, \phi^{\mathcal{PM}}_{Sem})\colon\ \mathcal{P} \to \mathcal{M}$ is a language morphism.*

Proof. **(Outline)** We need to show that $\phi^{\mathcal{PM}}_{Syn}$ and $\phi^{\mathcal{PM}}_{Sem}$ are behavioral refinements and that the diagram

$$\begin{array}{ccc} (\Sigma^{\mathcal{P}}_{Syn}, \emptyset) & \xrightarrow{\mathcal{P}} & (\Sigma^{\mathcal{P}}_{Sem}, E^{\mathcal{P}}_{Sem}) \\ \Big\downarrow{\scriptstyle \phi^{\mathcal{PM}}_{Syn}} & & \Big\downarrow{\scriptstyle \phi^{\mathcal{PM}}_{Sem}} \\ (\Sigma^{\mathcal{M}}_{Syn}, \emptyset) & \xrightarrow[\mathcal{M}]{} & (\Sigma^{\mathcal{M}}_{Sem}, E^{\mathcal{M}}_{Sem}) \end{array}$$

commutes. $\phi^{\mathcal{PM}}_{Syn}$ is a behavioral refinement trivially because $E^{\mathcal{P}}_{Syn} = \emptyset$. For $\phi^{\mathcal{PM}}_{Sem}$ we need to show that (*) $E^{\mathcal{M}}_{Sem} \models \phi^{\mathcal{PM}}_{Sem}(e)$, for all $e \in E^{\mathcal{P}}_{Sem}$ holds. Commutativity follows from $\phi^{\mathcal{PM}}_{Sem}{\restriction}_{\Sigma^{\mathcal{P}}_{Syn}} = \phi^{\mathcal{PM}}_{Syn}$.

The proof for (*) has been implemented in BOBJ. The proof is broken down into fourteen sub-proofs, one for each equation in $E^{\mathcal{P}}_{Sem}$. In order to accomplish these proofs we had to introduce five lemmas of which the first two are the most interesting, since these could only be proven using coinductive reasoning; strict satisfaction did not hold here. It is interesting to note that even in this fairly simple example we had to resort to behavioral equivalence rather than strict satisfaction in order to prove the compiler correct. The full BOBJ proof score is included in the appendix. The score breaks down into four major parts: the source language specification, the target language specification, the specification of the syntactic and semantic signature morphisms, and finally the proof of (*) above.

6 Related Work

Our work has its root in the algebraic approach to compiler correctness explored in the '60s and '70s [22, 4, 23, 29]. Here the source language syntax is seen as an initial algebra and in order to prove the correctness of the translation from the source language to the target language a homomorphism needs to be found between the source and target language semantic algebras. If such a homomorphism exists, initiality guarantees that the diagram representing the translation commutes, i.e., the translation is correct. Unfortunately, it is very difficult to find a homomorphism between the semantic algebras even for very similar languages.

Our approach to language specification and translation shares a number of similarities with the approaches advocated in [30, 3, 21]. In particular, rather than defining programming languages as algebras we define them as theories with interpretations or theory morphisms as translations between the languages. However, all of these approaches advocate strict satisfaction and therefore are limited in terms of the language translations they can prove correct.

The work most similar to ours is the work by da Silva [6]. This paper approaches the problem of compiler correctness by showing that the semantic specifications of two programming languages are equivalent under the observable portions of the theories given a particular mapping between the language syntax. However, instead of using first-order equational theories to define the semantics of the programming languages, da Silva uses *Relational Specification*. Currently, this formalism does not have any machine support.

An approach which is algebraic in its nature but conceptually quite different from ours is the work by Sampaio [26]. This work builds on an algebraic theory of refinement. Given a specification of the source language and a specification of the target language, as well as a set of transformation rules of how the source language should be transformed into the target language; it is necessary to show

that the transformation rules hold within a the theory of refinement. Once the correctness of the code transformation rules has been established they can be used to incrementally rewrite the source code into the target code, i.e., compile it. The difficulty here is establishing the correctness of the refinement theory.

7 Conclusions

Based on the intuition that a user will perceive a compiler as correct iff it preserves visible behavior, we developed the notions of behavioral language specification and language morphism based on a notion of hidden logic we find appropriate in this setting. Our model theoretic interpretations of these language specification constructs formalize precisely our intuition on visible behavior and compiler correctness.

We used these results to prove a simple compiler correct. It is interesting to note that even with the straightforward languages we chose, strict satisfaction was too restrictive to prove the compiler correct. Therefore, our framework based on hidden logic provides us with a correctness argument that allows us to prove compilers correct not possible in the strict setting.

We are currently undertaking a much more ambitious project than the translation of the simple assignment language described here. The source language consists of function calls, loop constructs, conditionals, and assignments. The target language is an abstract stack machine. The current behavioral specifications of these two languages consists of roughly 1000 lines of BOBJ code (one the largest behavioral specification we are aware of) and we are in the process of constructing a behavioral correctness proof using the framework developed here. Details can be found at the following web site:

`http://ase.arc.nasa.gov/grosu/download/bc.bob`

Acknowledgements We thank Kai Lin from UCSD for providing us with the latest versions of BOBJ and being extremely responsive when problems arose.

References

1. G. Bernot, M. Bidoit, and T. Knapik. Observational specifications and the indistinguishability assumption. *Theoretical Comp. Science*, 139(1-2):275–314, 1995.
2. M. Bidoit and R. Hennicker. Behavioral theories and the proof of behavioral properties. *Theoretical Computer Science*, 165(1):3–55, 1996.
3. M. Broy, M. Wirsing, and P. Pepper. On the algebraic definition of programming languages. *ACM Trans. on Prog. Lang. and Systems*, 9(1):54–99, January 1987.
4. R. Burstall and P. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969. Edinburgh University Press.
5. S. Buss and G. Roşu. Incompleteness of behavioral logics. In *Proceeding of CMCS'00*, volume 33 of *ENTCS*, pages 61–79. Elsevier, 2000.
6. F. da Silva. On observational equivalence and compiler correctness. In *Proceedings ICCI'94*, International Conf. on Computing and Information, pages 17–34, 1994.
7. R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, volume 6.

8. R. Diaconescu and K. Futatsugi. Behavioral coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 6(1):74–96, 2000.
9. J. Goguen. Types as theories. In *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991.
10. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992.
11. J. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In *Proceedings of WADT*, volume 785 of *LNCS*. Springer, 1994.
12. J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings of Automated Software Engineering 2000*, pages 123–131. IEEE, 2000.
13. J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.
14. J. Goguen and G. Malcolm. A hidden agenda. *Theoretical Computer Science*, 245(1):55–101, 2000.
15. J. Goguen and G. Roşu. Hiding more of hidden algebra. In *Proceeding of FM'99*, volume 1709 of *LNCS*, pages 1704–1719. Springer, 1999.
16. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.
17. L. Hamel. *Behavioural Verification and Implementation of an Optimizing Compiler for OBJ3*. PhD thesis, University of Oxford, 1996.
18. R. Hennicker. Context induction: a proof principle for behavioral abstractions. *Formal Aspects of Computing*, 3(4):326–345, 1991.
19. R. Hennicker and M. Bidoit. Observational logic. In *Proceedings of AMAST'98*, volume 1548 of *LNCS*, pages 263–277. Springer, 1999.
20. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.
21. B. Levy. *An Approach to Compiler Correctness Using Interpretation Between Theories*. PhD thesis, Comp. Sci. Lab., Aerospace Corporation, El Segundo, CA, 1986.
22. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of the Symposium in Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 33–41. 1967.
23. F. L. Morris. Advice on structuring compilers and proving them correct. In *Proceedings of POPL'73*, pages 144–152. ACM, 1973.
24. G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000. `http://ase.arc.nasa.gov/grosu/phd-thesis.ps`.
25. G. Roşu and J. Goguen. Hidden congruent deduction. In *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *LNAI*. Springer, 2000.
26. A. Sampaio. *An Algebraic Approach to Compiler Design*. PhD thesis, Oxford University, 1993.
27. J. Stoy. *Denotational Semantics of Programming Languages: The Scott-Strachey Approach to Programming Language Theory*. MIT, 1977.
28. C. Strachey. Towards a formal semantics. In Steel, editor, *Formal Language Description Languages*, pages 198–220. North-Holland, 1966.
29. J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. volume 71 of *LNCS*. Springer-Verlag, 1979.
30. M. Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980.

A Proof Score

```
*** ***************************************************
***> the assignment language
*** ***************************************************
*** define a variable name space
bth VAR is sort Var .
  ops u v w x y z temp : -> Var .
end

*** syntax specification
bth PROG-LANG-SYN is pr VAR .
  sorts Expr Stmt PrgmBody Prgm .
  subsort Var < Expr .  subsort Stmt < PrgmBody .
  op 0     : -> Expr .
  op S_    : Expr -> Expr [prec 10] .
  op _+_   : Expr Expr -> Expr [assoc comm prec 20] .
  op _:=_; : Var Expr -> Stmt .
  op __    : PrgmBody PrgmBody -> PrgmBody [assoc] .
  op  _!   : Expr -> Prgm .
  op __!   : PrgmBody Expr -> Prgm .
end

*** semantic specification
*** break out the signature so we can use it
*** to define operations without the equations.
bth PROG-LANG-SEM-SIGN is pr PROG-LANG-SYN .
  pr NAT .
  sorts Pair Env .  subsort Pair < Env .
  op [_,_]   : Var Nat -> Pair .
  op init    : -> Env .
  op anyenv  : -> Env .
  op __      : Env Env -> Env [assoc id: init] .
  op _[_<-_] : Env Var Nat -> Env .
  op _[_]    : Expr Env -> Nat .
  op _[_]    : PrgmBody Env -> Env .
  op _[_]    : Prgm Env -> Nat .
  op [_]     : Prgm -> Nat .
end

*** now the equations
bth PROG-LANG-SEM is pr PROG-LANG-SEM-SIGN .
  var St : Stmt .  var Pb : PrgmBody .  var P : Prgm .
  vars V V' : Var .  var Env : Env .  vars E E' : Expr .
  vars N N' : Nat .
  eq init[V <- N] = [V,N] .
  eq ([V',N'] Env)[V <- N] =
          if V == V' then [V,N] Env
          else [V',N'] (Env[V <- N]) fi .
  eq 0[Env] = 0 .
  eq (S E)[Env] = 1 + E[Env] .
  eq (E + E')[Env] = E[Env] + E'[Env] .
  eq V[init] = 0 .
  eq V[([V',N'] Env)] =
        if V == V' then N' else V[Env] fi .
  eq (V := E;)[Env] = Env[V <- E[Env]] .
  eq (St Pb)[Env] = Pb[St[Env]] .
  eq (E !)[Env] = E[Env] .
  eq (Pb E !)[Env] = E[Pb[Env]] .
  eq [P] = P[anyenv] .
end

cobasis PROG-LANG-COBASIS is
  op [_] : Prgm -> Nat .
end

*** ***************************************************
***> the machine language
*** ***************************************************
*** syntax specification
bth MACH-SYN is pr VAR .
  sorts Byte Instr Code .
  op o       : -> Byte .
  op s_      : Byte -> Byte .
  op loada_  : Byte -> Instr .
  op loada_  : Var -> Instr .
  op inca    : -> Instr .
  op null    : -> Instr .
  op adda_   : Byte -> Instr .
  op adda_   : Var -> Instr .
  op stora_  : Var -> Instr .
  op _;  : Instr -> Code .
  op __  : Code Code -> Code [assoc] .
end

*** semantic specification
bth MACH-SEM is pr MACH-SYN .  pr NAT .
  sorts State Location Mem .   subsort Location < Mem .
  vars C C' : Code .  var S : State .  var B : Byte .
  vars V V' : Var .  var Mem : Mem . var I : Instr .
  vars N N' : Nat .
  ops start anystate  : -> State .
  ops fresh anymem    : -> Mem .
  ops acc pcnt  : State -> Nat .
  op mem      : State -> Mem .
  op zeroAcc  : Mem -> State .
  op _,_    : Var Nat -> Location .
  op __       : Mem Mem -> Mem [assoc id: fresh] .
  op _[_<-_]  : Mem Var Nat -> Mem .
  op _[_]     : Mem Var -> Nat .
  op [_]      : Byte -> Nat .
  op _[_]     : Code State -> State .
  op [_]      : Code -> Nat .
  eq fresh[V <- N] = V,N .
  eq (V',N' Mem)[V <- N] =
     if V == V' then V,N Mem else V',N' (Mem[V <- N]) fi .
  eq fresh[V] = 0 .
  eq (V',N' Mem)[V] = if V == V' then N' else Mem[V] fi .
  eq [o] = 0 .
  eq [s B] = 1 + [B] .
  eq (C C')[S] = C'[C[S]] .
  eq acc(start)          = 0 .
  eq acc(loada B;  [S]) = [B] .
  eq acc(loada V;  [S]) = mem(S)[V] .
  eq acc(inca;     [S]) = acc(S) + 1 .
  eq acc(null;     [S]) = acc(S) .
  eq acc(adda B;   [S]) = acc(S) + [B] .
  eq acc(adda V;   [S]) = acc(S) + mem(S)[V] .
  eq acc(stora V;  [S]) = acc(S) .
  eq mem(start)          = fresh .
  eq mem(loada B;  [S]) = mem(S) .
  eq mem(loada V;  [S]) = mem(S) .
  eq mem(inca;     [S]) = mem(S) .
  eq mem(null;     [S]) = mem(S) .
  eq mem(adda B;   [S]) = mem(S) .
  eq mem(adda V;   [S]) = mem(S) .
  eq mem(stora V;  [S]) = mem(S)[V <- acc(S)] .
  eq pcnt(I; [S]) = 1 + pcnt(S) .
  eq acc(zeroAcc(Mem)) = 0 .
  eq mem(zeroAcc(Mem)) = Mem .
  eq pcnt(zeroAcc(Mem)) = 0 .
  eq [C] = acc(C[anystate]) .
end

cobasis MACH-COBASIS is
  op [_] : Code -> Nat .
  op acc : State -> Nat .
  op mem : State -> Mem .
end

*** ***************************************************************
***> translation
*** ***************************************************************
*** define the signature morphism for the syntactic theories.
bth PHI-SYN is pr PROG-LANG-SYN .  pr MACH-SYN .
  op phiSyn : Prgm -> Code .
  op phiSyn : PrgmBody -> Code .
  op phiSyn : Expr -> Code .
  var V : Var .  vars E E' : Expr .  var St : Stmt.
  var Pb : PrgmBody .
  eq phiSyn(E !) = loada o; phiSyn(E) .
  eq phiSyn(Pb E !) = phiSyn(Pb) (loada o;) phiSyn(E) .
  eq phiSyn(V := E;) = (loada o;) phiSyn(E) (stora V;) .
  eq phiSyn(St Pb) = phiSyn(St) phiSyn(Pb) .
  eq phiSyn(V) = adda V; .
  eq phiSyn(0) = null; .
  eq phiSyn(S E) = phiSyn(E) inca; .
  eq phiSyn(E + E') = phiSyn(E) phiSyn(E') .
end

*** define the signature morphism for the semantic theories
bth PHI-SEM is pr PROG-LANG-SEM-SIGN .  pr MACH-SEM .
  pr PHI-SYN .
  var V : Var .  var N : Nat .  vars Env Env' : Env .
  var Pb : PrgmBody . var P : Prgm . var E : Expr .
  op phiSem : Nat -> Nat . *** identity
  op phiSem : Pair -> Location .
  op phiSem : Env -> Mem .
  eq phiSem(P[Env]) = acc(phiSyn(P)[zeroAcc(phiSem(Env))]) .
  eq phiSem([P]) = [phiSyn(P)] .
  eq phiSem(E[Env]) = acc(phiSyn(E)[zeroAcc(phiSem(Env))]) .
  eq phiSem([V,N]) = V,N .
  eq phiSem(init) = fresh .
  eq phiSem(anyenv) = anymem .
  eq phiSem(Env Env') = phiSem(Env) phiSem(Env') .
  eq phiSem(Env[V <- N]) = phiSem(Env)[V <- phiSem(N)] .
  eq phiSem(Pb[Env]) = mem(phiSyn(Pb)[zeroAcc(phiSem(Env))]) .
end
```

```
*** ***********************************************
***> proof that source equations hold in target
*** ***********************************************
*** this is the proof the the semantic signature
*** morphism is indeed a refinement.

*** we next prove five lemmas.
*** once a lemma is proved, it is added as an equation because
*** it is useful to prove the others
open PHI-SEM .
var State : State .  var Pb : PrgmBody .  var St : Stmt .
var V : Var .  vars E E' : Expr .
set cobasis MACH-COBASIS

*** lemma 1: coinduction
cred zeroAcc(fresh) == start .
 eq zeroAcc(fresh) = start .

*** lemma 2: coinduction
cred loada o; [State] == zeroAcc(mem(State)) .
 eq loada o; [State] = zeroAcc(mem(State)) .

*** lemma 3: needed only locally, to prove the next one
red phiSyn(V := E;)[zeroAcc(mem(State))] ==
    phiSyn(V := E;)[State] .
 eq phiSyn(St)[zeroAcc(mem(State))] = phiSyn(St)[State] .

*** lemma 4: induction on PrgmBody
red phiSyn(St Pb)[zeroAcc(mem(State))] == phiSyn(St Pb)[State] .
 eq phiSyn(Pb)[zeroAcc(mem(State))] = phiSyn(Pb)[State] .

*** lemma 5: next two by induction on Expr
ops e e' : -> Expr .
eq mem(phiSyn(e)[State]) = mem(State) .
eq mem(phiSyn(e')[State]) = mem(State) .
red mem(phiSyn(S e)[State]) == mem(State) .
red mem(phiSyn(e + e')[State]) == mem(State) .
 eq mem(phiSyn(E)[State]) = mem(State) .
ceq acc(phiSyn(e)[State]) =
    acc(phiSyn(e)[zeroAcc(mem(State))]) + acc(State)
      if acc(State) =/= 0 .
ceq acc(phiSyn(e')[State]) =
    acc(phiSyn(e')[zeroAcc(mem(State))]) + acc(State)
      if acc(State) =/= 0 .
red acc(phiSyn(0)[zeroAcc(mem(State))]) + acc(State) ==
    acc(phiSyn(0)[State]) .
red acc(phiSyn(S e)[zeroAcc(mem(State))]) + acc(State) ==
    acc(phiSyn(S e)[State]) .
red acc(phiSyn(e + e')[zeroAcc(mem(State))]) + acc(State) ==
    acc(phiSyn(e + e')[State]) .
close

*** therefore, we can safely add these lemmas as axioms
*** we create a theory so that we can easily include
*** these lemmas in our proofs below.
bth LEMMAS is pr PHI-SEM .
  var E : Expr .  var State : State .  var Pb : PrgmBody .
  var V : Var . var N : Nat .
  eq zeroAcc(fresh) = start .
  eq (loada o; [State]) = zeroAcc(mem(State)) .
  eq phiSyn(Pb)[zeroAcc(mem(State))] = phiSyn(Pb)[State] .
  eq mem(phiSyn(E)[State]) = mem(State) .
  ceq acc(phiSyn(E)[State]) =
      acc(phiSyn(E)[zeroAcc(mem(State))]) + acc(State)
        if acc(State) =/= 0 .
  eq V, phiSem(N) = V,N .  *** phiSem is the identity on integers
end

*** we now prove that each equation in the
*** semantic theory of PROG-LANG holds
*** in the target semantic theory.
open LEMMAS .
vars V V' : Var .  vars N N' : Nat .
vars Env Env' Env'' : Env .
vars E E' : Expr .  var State : State .
var St : Stmt .  var Pb : PrgmBody .  var P : Prgm .

*** attribute of "op __: Env Env -> Env" : associativity
red phiSem(Env) (phiSem(Env') phiSem(Env'')) ==
    (phiSem(Env) phiSem(Env')) phiSem(Env'') .

*** attribute of "op __: Env Env -> Env" : identity of init
red phiSem(Env) phiSem(init) == phiSem(Env) .
red phiSem(init) phiSem(Env) == phiSem(Env) .

*** Equation 1:
*** init[V <- N] = [V,N] .
red phiSem(init[V <- N]) == phiSem([V,N]) .

*** Equation 2:
*** ([V',N'] Env)[V <- N] =
***    if V == V' then [V,N] Env else [V',N'] (Env[V <- N]) fi .
*** case 1 : V == V'
red phiSem(([V,N'] Env)[V <- N]) == phiSem([V,N]Env) .
*** case 2 : V =/= V'
red phiSem(([V',N'] Env)[V <- N]) == phiSem([V',N'] (Env[V <- N])) .

*** Equation 3:
*** 0[Env] = 0 .
red phiSem(0[Env]) == 0 .

*** Equation 4:
*** (S E)[Env] = 1 + E[Env] .
red phiSem((S E)[Env]) == 1 + phiSem(E[Env]) .

*** Equation 5:
*** (E + E')[Env] = E[Env] + E'[Env] .
red phiSem((E + E')[Env]) == phiSem(E[Env]) + phiSem(E'[Env]) .

*** Equation 6:
*** V[init] = 0 .
red phiSem(V[init]) == 0 .

*** Equation 7:
*** V[([V',N'] Env)] = if V == V' then N' else V[Env] fi .
*** case 1 : V == V'
red phiSem(V[([V,N'] Env)]) == N' .
*** case 2 : V =/= V'
red phiSem(V[([V',N'] Env)]) == phiSem(V[Env]) .

*** Equation 8:
*** (V := E;)[Env] = Env[V <- E[Env]] .
red phiSem((V := E;)[Env]) == phiSem(Env[V <- E[Env]]) .

*** Equation 9:
*** (St Pb)[Env] = Pb[St[Env]] .
red phiSem((St Pb)[Env]) == phiSem(Pb[St[Env]]) .

*** Equation 10:
*** (E !)[Env] = E[Env] .
red phiSem((E !)[Env]) == phiSem(E[Env]) .

*** Equation 11:
*** (Pb E !)[Env] = E[Pb[Env]] .
red phiSem((Pb E !)[Env]) == phiSem(E[Pb[Env]]) .

*** Equation 12:
*** [P] = P[env] .
*** assumption:
eq anystate = zeroAcc(anymem) .
red phiSem([P]) == phiSem(P[anyenv]) .
close
```