Implementing the Flow Logic for Carmel

Authors	: René Rydhof Hansen
Date	: November 13, 2002
Number	: SECSAFE-IMM-004-1.0
Classification	: Public

1 Introduction

In [4] a control flow analysis for Carmel is specified and proved correct with respect to the semantics as defined in [9]. The Carmel language is described in detail in [6, 9]. Extensions to the basic control flow analysis are discussed in [3]; we shall not go into further detail with the control flow analysis or the extensions of it here. In this paper we show how the flow logic specification given in [4] systematically can be transformed into a specification for generating constraints in the Alternation-free Least Fixed-point logic (ALFP). Constraints over this language can be solved using the techniques described in [8, 7], in particular they can be solved using the "Succinct Solver" of Nielson and Seidl.

The transformation is, deliberately, kept in a "high-level" style that is very close to that of the Flow Logic specification. This is in part to facilitate the proof of semantic correctness for the transformation and also in part to simplify the transformation process. This comes at the price of lower efficiency, with respect to both time and space, when solving the generated constraints. In Section 6 a number of strategies for optimising the constraint generation are discussed.

The remainder of the paper is structured as follows. Section 2 introduces the Alternation-free Least Fixed-Point (ALFP) logic in more detail. Section 3 details how the abstract domains are represented as ALFP constraints and Section 4 then shows how the Flow Logic specification is converted into ALFP constraints; this includes a full specification of the constraint generation for all the supported instructions. Semantic correctness of the conversion from Flow Logic to ALFP is discussed in Section 5. A number of optimisation strategies regarding the generated constraints are described in Section 6 and then in Section 7 we give an overview of a prototype implementation of the constraint generator. Conclusions and Future Work can be found in Section 8.

2 Alternation-free Least Fixed-Point logic

The fundamental idea in implementing the analysis of [4] is to convert the flow logic clauses to clauses in the *Alternation-free Least Fixed-Point* (ALFP) logic, because efficient means for finding solutions to ALFP clauses are well known, such as the *Succinct Solver* technology, cf. [7, 8]. In the following we give a brief introduction to the ALFP logic.

2.1 The constraint Language

Formulae in ALFP consists of clauses of the following form:

$$\begin{array}{rcl}
\mathsf{pre} & ::= & R(x_1,\dots,x_k) \mid \mathsf{pre}_1 \wedge \mathsf{pre}_2 \mid \mathsf{pre}_1 \vee \mathsf{pre}_2 \mid \exists x : \mathsf{pre} \\
\\
\mathsf{clause} & ::= & R(x_1,\dots,x_k) \mid 1 \mid \mathsf{clause}_1 \wedge \mathsf{clause}_2 \\
& \mid & \mathsf{pre} \Rightarrow \mathsf{clause} \mid \forall x : \mathsf{clause}
\end{array}$$

where R is a k-ary relation symbol for $k \geq 1$ and $x_1, \dots$ denote variables while $\mathbf{1}$ is the always true clause.

In the following let t be a pre-condition or a clause. Then for a given universe, $\mathcal{U}$, of atomic values, and interpretations ρ for relation symbols and σ for free variables, the satisfaction relation for pre-conditions and clauses, $(\rho,\sigma) \models_{\mathrm{ALFP}} t$, can be defined as follows:

$$\begin{array}{lll}
(\rho,\sigma) \models_{\mathrm{ALFP}} \mathbf{1} & \text{iff} & \text{true} \\
(\rho,\sigma) \models_{\mathrm{ALFP}} R(x_1,\dots,x_k) & \text{iff} & (\sigma x_1,\dots,\sigma x_k) \in \rho R \\
(\rho,\sigma) \models_{\mathrm{ALFP}} \exists x : \mathsf{pre} & \text{iff} & (\rho,\sigma[x \mapsto a]) \models_{\mathrm{ALFP}} \mathsf{pre} \quad \text{for some } a \in \mathcal{U} \\
(\rho,\sigma) \models_{\mathrm{ALFP}} \forall x : t & \text{iff} & (\rho,\sigma[x \mapsto a]) \models_{\mathrm{ALFP}} t \quad \text{for all } a \in \mathcal{U} \\
(\rho,\sigma) \models_{\mathrm{ALFP}} t_1 \wedge t_2 & \text{iff} & (\rho,\sigma) \models_{\mathrm{ALFP}} t_1 \text{ and } (\rho,\sigma) \models_{\mathrm{ALFP}} t_2 \\
(\rho,\sigma) \models_{\mathrm{ALFP}} \mathsf{pre}_1 \vee \mathsf{pre}_2 & \text{iff} & (\rho,\sigma) \models_{\mathrm{ALFP}} \mathsf{pre}_1 \text{ or } (\rho,\sigma) \models_{\mathrm{ALFP}} \mathsf{pre}_2 \\
(\rho,\sigma) \models_{\mathrm{ALFP}} \mathsf{pre} \Rightarrow \mathsf{cl} & \text{iff} & (\rho,\sigma) \models_{\mathrm{ALFP}} \mathsf{cl} \text{ whenever } (\rho,\sigma) \models_{\mathrm{ALFP}} \mathsf{pre}
\end{array}$$

For a given interpretation, σ, of variable symbols we call an interpretation, ρ, of relation symbols a *solution* to a clause clause if indeed $(\rho,\sigma) \models_{\mathrm{ALFP}} \mathsf{clause}$.

2.2 Solving the Constraints

Using the techniques of [7] it is possible to efficiently find solutions to given clauses. An implementation, called Succinct Solver, using these and other advanced techniques has been made by Nielson and Seidl and is described in [8].

3 Representing the Abstract Domains

This section is devoted to showing how the abstract domains used in the control flow analysis can be represented in the constraint language.

Basic values. For the pure control flow analysis, integers are modeled as a single token: INT. For a simple data flow analysis, this can easily be extended to a single token for each constant integer, v, occurring in the program: INT_v and a top-element to represent “unknown”: $\mathsf{INT}_\top$.

Object references modeled simply as classnames: an object reference to a class *classname* is represented by $\mathsf{cl\_classname}$. Similarly arrays of type t are represented as $\mathsf{ar\_t}$.

Stacks. In order to model the abstract stack we use a quarternary relation, S, relating adresses and stack positions to values, thus the clause

$$\mathsf{S}(m_0, pc_0, [3], \mathsf{INT})$$

is intended to mean that the abstract stack at address (m_0, pc_0) contains an integer value at stack position three.

Since we must be able to manipulate and calculate stack positions directly within the clauses, stack positions must be represented explicitly:

$$\begin{array}{rcl} [0] & = & \texttt{zero} \\ [n+1] & = & \texttt{suc(}[n]\texttt{)} \end{array}$$

Instead of constructing numbers, as described above, special constants could be used to directly encode numbers, eg. `int_7` for the number seven; however, such an approach also would require that tables for doing arithmetic are encoded explicitly.

Since only stack positions, and not eg. local variable indices, are manipulated or calculated directly within the clauses, only these need to be represented using the above.

We can now model an abstract stack, $\hat{S}(m_0, pc_0) = A_1 :: \cdots :: A_n$, at address (m_0, pc_0) where $A_i = \{a_i^1, \ldots, a_i^{j_i}\}$ as follows:

$$\begin{array}{rcl} \mathsf{S}(m_0, pc_0, [0], a_1^1)\wedge & \ldots & \wedge\mathsf{S}(m_0, pc_0, [0], a_1^{j_1})\wedge \\ & \vdots & \\ \mathsf{S}(m_0, pc_0, [n-1], a_n^1)\wedge & \ldots & \wedge\mathsf{S}(m_0, pc_0, [n-1], a_n^{j_n}) \end{array}$$

Thus, the top of the stack is at position zero, [0], with the rest of the stack in the following positions.

Due to the way the current version of the solver handles terms, such as stack positions, it is necessary to calculate stack positions "in advance", ie. in the precondition, using unification. Thus

$$\forall i\colon \forall a\colon \mathsf{S}(m_0, pc_0, [i], a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, [i+1], a)$$

should be written

$$\forall i\colon \forall a\colon \forall y\colon (y = [i+1]) \wedge \mathsf{S}(m_0, pc_0, [i], a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, y, a)$$

Local Heap. The local heap is an array containing the values for the local variables. As for stacks a local heap is assigned to every address of the program. But unlike for stacks, we do not need to manipulate the index or position of local variables, and therefore we do not need a special representation for these but can use simple tokens: $\mathsf{var\_3}$ to mean the local variable at index 3.

Again we use a quarternary relation to relate addresses and variable index to values:

$$\mathsf{L}(m_0, pc_0, \mathsf{var\_2}, \mathsf{INT})$$

The above models that the local variable with index 2, at address (m_0, pc_0) has an integer value.

Heap and static Heap. The heap is represented as a ternary relation that relates object references and field id's to values, if for instance $\hat{H}(\text{Ref }\sigma)(f.\text{id}) = v$ then

$$\mathsf{H}(\mathsf{cl\_}\sigma, f.\text{id}, v)$$

Since the array is used for both objects and arrays, we use a "dummy" value, ARRAY, to differentiate between the two uses. Thus if $\hat{H}(\mathrm{Ref}\ (\texttt{array}\ t)) = \{\mathsf{INT}\}$ then

$$\mathsf{H}(\mathsf{ar\_t}, \mathsf{ARRAY}, \mathsf{INT})$$

The static heap is represented in a similar manner, except that no references are needed, only field identifiers, cf. [9]. As an example, if $\hat{K}(f.\mathrm{id}) = v$ then

$$\mathsf{K}(f.\mathrm{id}, v)$$

Method Lookup. In the semantics a function, *methodLookup*, is defined that given a method identifier and a class it looks up the implementation of the method, based on the class hierarchy. This is modeled in the expected manner in the constraint language, for instance

$$m_v = \mathit{methodLookup}(m.\mathrm{id}, \sigma)$$

is translated into

$$\mathsf{ML}(m.\mathrm{id}, \mathsf{cl\_}\sigma, mv)$$

The constraints defining the relation ML are derived directly from the class hierarchy.

End Tokens. The special tokens occurring in addresses indicating the end of a method, used for transferring return values from methods, are represented as constants in the constraint language. A special relation then relates such constants to the method whose end token they represent:

$$\mathsf{END}(m, \mathit{end\_m})$$

meaning that the constant *end_m* now represents the end token for the method represented by m. The relation can then be used for looking up the particular end token for a given method. In certain situations the end token need not be looked up but can be computed directly; in such cases we write END_m for the token belonging to method m.

Like the ML relation, the END relation depends trivially on the structure of programs and therefore it is *defined* from the outset rather than solved for.

4 Generating Constraints

In this section we show how to systematically transform the flow logic specification of the analysis, cf. [4], into another flow logic specification based on ALFP. Using the Succinct Solver technology, cf. Section 2.2, we can efficiently find solutions to ALFP clauses and thereby obtain an implementation of the control flow analysis.

We first present and discuss the new specification in detail for a few representative instructions (Section 4.1), followed by the full specification (Section 4.2).

4.1 A Few Interesting Clauses

4.1.1 The `push` Instruction

For the `push`-instruction the flow logic specification is given as follows:

$$\begin{aligned}&(\hat{K},\hat{H},\hat{L},\hat{S}) \models (m_0, pc_0) : \texttt{push}\ t\ \ v\\ &\quad \text{iff}\quad \{v\} :: \hat{S}(m_0,pc_0) \sqsubseteq \hat{S}(m_0,pc_0+1)\\ &\qquad\quad\ \hat{L}(m_0,pc_0) \sqsubseteq \hat{L}(m_0,pc_0+1)\end{aligned}$$

This entails that the value being pushed, v, should be put on top of the stack, ie. in position zero, in the stack for the next instruction at $pc_0 + 1$:

$$\mathsf{S}(m_0, pc_0 + 1, [0], \mathsf{INT})$$

Note that the actual value, v, is replaced by a single token INT as we do not track the actual values in the control flow analysis.

Furthermore, the entire stack for the current instruction (before the `push`-instruction is executed) must be copied to the instruction at $pc_0 + 1$ but moved one stack position down, eg. what was in position [0] should be copied to position [1], this is accomplished by the clause below:

$$\forall i{:}\, \forall a{:}\, \forall y{:}\, (y = [i+1]) \wedge \mathsf{S}(m_0, pc_0, [i], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, y, a)$$

While the above clause is a very straightforward translation of the Flow Logic specification, it is quite expensive in terms of both time and space. In Section 6 a number of optimisation strategies for improving this situation are discussed.

Finally, the local variables are untouched and thus copied forward unchanged:

$$\forall x \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$$

putting all of the above clauses together, we arrive at:

$$\begin{aligned}&(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{push}\ t\ \ v \qquad \text{iff}\\ &\quad \mathsf{S}(m_0, pc_0+1, [0], \mathsf{INT})\\ &\quad \forall i : \forall a : \forall y : (y = [i+1]) \wedge \mathsf{S}(m_0, pc_0, [i], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, y, a)\\ &\quad \forall x{:}\, \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)\end{aligned}$$

4.1.2 The `store` Instruction

The `store`-instruction is analysed as follows

$$\begin{aligned}&(\hat{K},\hat{H},\hat{L},\hat{S}) \models (m_0, pc_0) : \texttt{store}\ t\ \ x\\ &\quad \text{iff}\quad A :: X \lhd \hat{S}(m_0,pc_0) :\\ &\qquad\quad\ X \sqsubseteq \hat{S}(m_0,pc_0+1)\\ &\qquad\quad\ A \sqsubseteq \hat{L}(m_0,pc_0+1)(x)\\ &\qquad\quad\ \hat{L}(m_0,pc_0) \sqsubseteq_{\{x\}} \hat{L}(m_0,pc_0+1)\end{aligned}$$

Intuitively $A :: X \lhd \hat{S}(m_0, pc_0)$ is a binding operator, in that it binds whatever is on top of the abstract stack at the current address to the variable A, which can then be referenced in a later clause. For a more thorough explanation of the specification and the notation used see [4].

Following the above specification we see that the values on top of the stack should be copied to the local variable x, this gives rise to the following constraints:

$$\forall a : \mathsf{S}(m_0, pc_0, [0], a) \;\Rightarrow\; \mathsf{L}(m_0, pc_0 + 1, \mathsf{var\_x}, a)$$

Note that, as decribed previously, the local variable x is represented as the token $\mathsf{var\_x}$.

Next the remainder of the stack must be moved one place, as the top of the stack is popped after having been copied. This is formulated in the constraint language using the following clause:

$$\forall i : \forall a : \forall y : (y = [i+1]) \land \mathsf{S}(m_0, pc_0, y, a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, [i], a)$$

Having moved the stack into place, we now must copy all the local variables onwards, except the variable x which has just been updated and thus we need not copy the old value for x forwards:

$$\forall y : \forall a : (y \neq \mathsf{var\_x}) \land \mathsf{L}(m_0, pc_0, y, a) \;\Rightarrow\; \mathsf{L}(m_0, pc_0 + 1, y, a)$$

Finally, combining the above we have the following constraints for the `store`-instruction

$$\begin{array}{l}(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{store}\ t\ x \qquad \text{iff} \\ \quad \forall a : \mathsf{S}(m_0, pc_0, [0], a) \;\Rightarrow\; \mathsf{L}(m_0, pc_0 + 1, \mathsf{var\_x}, a) \\ \quad \forall i : \forall a : \forall y : (y = [i+1]) \land \mathsf{S}(m_0, pc_0, y, a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, [i], a) \\ \quad \forall y : \forall a : (y \neq \mathsf{var\_x}) \land \mathsf{L}(m_0, pc_0, y, a) \;\Rightarrow\; \mathsf{L}(m_0, pc_0 + 1, y, a)\end{array}$$

4.1.3 The `putfield` Instruction

Storing values in instance fields is accomplished by the `putfield`-instruction. Based on the analysis of the instruction

$$\begin{array}{l}(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{putfield}\ f \\ \quad \text{iff} \quad A :: B :: X \lhd \hat{S}(m_0, pc_0) : \\ \qquad\quad \forall (\text{Ref } \sigma') \in B : A \sqsubseteq \hat{H}(\text{Ref } \sigma')(f.\text{id}) \\ \qquad\quad X \sqsubseteq \hat{S}(m_0, pc_0 + 1) \\ \qquad\quad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0 + 1)\end{array}$$

we see that the value on top of the stack is copied into the field pointed to by the reference found in the second position of the stack. Converting this to constraints we get

$$\forall r : \forall a : \mathsf{S}(m_0, pc_0, [1], r) \land \mathsf{S}(m_0, pc_0, [0], a) \;\Rightarrow\; \mathsf{H}(r, f.\text{id}, a)$$

Now the remainder of the stack, the original stack less the top two elements, should be copied onwards to the next instruction:

$$\forall y : \forall a : \forall i : y = [i+2] \land \mathsf{S}(m_0, pc_0, y, a) \;\Rightarrow\; \mathsf{S}(m_0, pc_0 + 1, [i], a)$$

Finally, none of the local variables were modified and should simply be copied onwards using the familiar clause:

$$\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \;\Rightarrow\; \mathsf{L}(m_0, pc_0 + 1, x, a)$$

Combining the above constraints we can formulate the clause for `putfield`:

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{putfield}\ f \qquad \text{iff} \\
\quad \forall r : \forall a : \mathsf{S}(m_0, pc_0, [1], r) \land \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{H}(r, f, a) \\
\quad \forall y : \forall a : \forall i : y = [i+2] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

4.1.4 The `invokevirtual` Instruction

Converting the specification for `invokevirtual` follows the same general pattern as above although it is a bit more involved. First we recall the flow logic specification for `invokevirtual`:

$$
\begin{array}{l}
(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models (m_0, pc_0) : \texttt{invokevirtual}\ m \\
\quad \text{iff} \quad A_1 :: \cdots :: A_{|m|} :: B :: X \lhd \hat{S}(m_0, pc_0) : \\
\qquad \forall (\text{Ref } \sigma') \in B : \\
\qquad\quad m_v = \mathit{methodLookup}(m.\text{id}, \sigma') \\
\qquad\quad \{(\text{Ref } \sigma')\} :: A_1 :: \cdots :: A_{|m|} \sqsubseteq \hat{L}(m_v, 1)[0..|m_v|] \\
\qquad\quad m.\text{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad T :: Y \lhd \hat{S}(m_v, \mathsf{END}_{m_v}) : \ T :: X \sqsubseteq \hat{S}(m_0, pc_0+1) \\
\qquad\quad m.\text{returnType} = \texttt{void} \Rightarrow \\
\qquad\qquad X \sqsubseteq \hat{S}(m_0, pc_0+1) \\
\qquad \hat{L}(m_0, pc_0) \sqsubseteq \hat{L}(m_0, pc_0+1)
\end{array}
$$

First the reference to the object where the invoked method resides is copied (as a self reference) to local variable 0 of the invoked method:

$$\forall r \forall mv : \mathsf{S}(m_0, pc_0, [|m|], r) \land \mathsf{ML}(m.\text{id}, r, mv) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_0}, r)$$

Next the parameters are transferred from the stack of the current method to the local variables of the invoked method:

$$
\begin{array}{l}
\forall r \forall mv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \land \mathsf{ML}(m.\text{id}, r, mv) \land \\
\quad \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_1}, a) \\
\qquad\qquad\qquad \vdots \\
\forall r \forall mv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \land \mathsf{ML}(m.\text{id}, r, mv) \land \\
\quad \mathsf{S}(m_0, pc_0, [|m|-1], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_|m|}, a)
\end{array}
$$

In case the method returns a value, that value should be put on top of the stack for the next instruction, and the rest of the current stack, less the arguments to the invoked method, is also copied forward. Thus if $m.\text{returnType} \neq$ `void` then the following constraints are generated:

$$
\begin{array}{l}
\forall y : \forall z : \forall a : \forall i : \\
\quad y = [i + |m| + 1] \land z = [i+1] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, z, a) \\
\forall r \forall mv \forall \mathit{endmv} \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \land \mathsf{ML}(m.\text{id}, r, mv) \land \\
\quad \mathsf{END}(mv, \mathit{endmv}) \land \mathsf{S}(mv, \mathit{endmv}, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [0], a)
\end{array}
$$

If on the other hand the invoked method does not return a value, then only the current stack, less the arguments to the invoked method, is copied forward. Thus if $m.\text{returnType} =$ `void` then the following constraints are generated:

$$\forall y : \forall a : \forall i : y = [i + |m| + 1] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a)$$

Finally, since the local variables of the invoking method are not modified, they are simply copied along as well:

$$\forall x \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$$

Putting it all together we arrive at:

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{invokevirtual}\ m$   iff
  $\forall r : \forall mv : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \Rightarrow$
    $\mathsf{L}(mv, 1, \mathsf{var\_0}, r)$
  $\forall r : \forall mv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
    $\mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_1}, a)$
                              $\vdots$
  $\forall r : \forall mv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
    $\mathsf{S}(m_0, pc_0, [|m| - 1], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_|m|}, a)$
  if $m$.returnType $\neq$ void then
    $\forall y : \forall z : \forall a : \forall i :$
      $y = [i + |m| + 1] \wedge z = [i + 1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, z, a)$
    $\forall r : \forall mv : \forall endmv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
      $\mathsf{END}(mv, endmv) \wedge \mathsf{S}(mv, endmv, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a)$
  if $m$.returnType $=$ void then
    $\forall y : \forall a : \forall i : y = [i + |m| + 1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [i], a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$
```

4.2 Full Specification

Below the full specification follows.

Stack manipulation.

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{push}\ t\ v$   iff
  $\mathsf{S}(m_0, pc_0 + 1, [0], \mathsf{INT})$
  $\forall i : \forall a : \forall y : (y = [i + 1]) \wedge \mathsf{S}(m_0, pc_0, [i], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, y, a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$
```

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{pop}\ n$   iff
  $\forall y : \forall i : \forall a : y = [i + n] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [i], a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$
```

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{dup}\ m\ n$   iff
  $\forall a : \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a)$
                              $\vdots$
  $\forall a : \mathsf{S}(m_0, pc_0, [n - 1], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [n - 1], a)$
  $\forall a : \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [n], a)$
                              $\vdots$
  $\forall a : \mathsf{S}(m_0, pc_0, [m - 1], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [(m - 1) + n], a)$
  $\forall y : \forall z : \forall i : \forall a : y = [i + n] \wedge z = [i + n + m] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow$
    $\mathsf{S}(m_0, pc_0 + 1, z, a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$
```

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{swap}\ m\ n \qquad \text{iff} \\
\quad \forall a : \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [n], a) \\
\qquad\qquad\qquad\qquad \vdots \\
\quad \forall a : \mathsf{S}(m_0, pc_0, [m-1], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [(m-1)+n], a) \\
\quad \forall a : \mathsf{S}(m_0, pc_0, [m], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [0], a) \\
\qquad\qquad\qquad\qquad \vdots \\
\quad \forall a : \mathsf{S}(m_0, pc_0, [(n-1)+m], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [n-1], a) \\
\quad \forall y : \forall z : \forall i : \forall a : y = [i+n+m] \wedge z = [i+n+m] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \\
\qquad \mathsf{S}(m_0, pc_0+1, z, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

Arithmetic operators.

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{numop}\ t\ \mathit{unop}\ t'_{opt} \qquad \text{iff} \\
\quad \mathsf{S}(m_0, pc_0+1, [0], \mathsf{INT}) \\
\quad \forall y : \forall z : \forall i : \forall a : y = [i+1] \wedge z = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \\
\qquad \mathsf{S}(m_0, pc_0+1, z, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{numop}\ t\ \mathit{binop}\ t'_{opt} \qquad \text{iff} \\
\quad \mathsf{S}(m_0, pc_0+1, [0], \mathsf{INT}) \\
\quad \forall y : \forall z : \forall i : \forall a : y = [i+2] \wedge z = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \\
\qquad \mathsf{S}(m_0, pc_0+1, z, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

Control flow.

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{goto}\ L \qquad \text{iff} \\
\quad \forall i : \forall a : \mathsf{S}(m_0, pc_0, [i], a) \Rightarrow \mathsf{S}(m_0, L, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, L, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{if}\ t\ \mathit{cmpop}\ \texttt{goto}\ L \qquad \text{iff} \\
\quad \forall y : \forall i : \forall a : y = [i+2] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a) \\
\quad \forall y : \forall i : \forall a : y = [i+2] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, L, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, L, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{if}\ t\ \mathit{cmpop}\ \mathit{nullCmp}\ \texttt{goto}\ L \qquad \text{iff} \\
\quad \forall y : \forall i : \forall a : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a) \\
\quad \forall y : \forall i : \forall a : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, L, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, L, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{lookupswitch}\ t\ (k_i\texttt{=>}L_i)_1^n,\ \texttt{default=>}L_0 \qquad \text{iff} \\
\quad \forall y : \forall i : \forall a : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, L_0, [i], a) \\
\qquad\qquad\qquad\qquad \vdots \\
\quad \forall y : \forall i : \forall a : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, L_n, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, L_0, x, a) \\
\qquad\qquad\qquad\qquad \vdots \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, L_n, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{tableswitch}\ t\ l\texttt{=>}(L_i)_1^n,\ \texttt{default=>}L_0 \qquad \text{iff} \\
\quad \forall y : \forall i : \forall a : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \ \Rightarrow\ \mathsf{S}(m_0, L_0, [i], a) \\
\qquad \vdots \\
\quad \forall y : \forall i : \forall a : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \ \Rightarrow\ \mathsf{S}(m_0, L_n, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, L_0, x, a) \\
\qquad \vdots \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, L_n, x, a)
\end{array}
$$

Local variables.

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{load}\ t\ x \qquad \text{iff} \\
\quad \forall a : \mathsf{L}(m_0, pc_0, \mathsf{var\_x}, a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, [0], a) \\
\quad \forall y : \forall i : \forall a : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, [i], a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, y, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{store}\ t\ x \qquad \text{iff} \\
\quad \forall a : \mathsf{S}(m_0, pc_0, [0], a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, \mathsf{var\_x}, a) \\
\quad \forall i : \forall a : \forall y : (y = [i+1]) \wedge \mathsf{S}(m_0, pc_0, y, a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, [i], a) \\
\quad \forall y : \forall a : (y \neq \mathsf{var\_x}) \wedge \mathsf{L}(m_0, pc_0, y, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, y, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{inc}\ t\ x\ c \qquad \text{iff} \\
\quad \forall i : \forall a : \mathsf{S}(m_0, pc_0, [i], a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

Object Language.

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{checkcast}\ t \qquad \text{iff} \\
\quad \forall i : \forall a : \mathsf{S}(m_0, pc_0, [i], a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{instanceof}\ t \qquad \text{iff} \\
\quad \mathsf{S}(m_0, pc_0+1, [0], \mathsf{INT}) \\
\quad \forall y : \forall a : \forall i : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, [i], a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, y, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{new}\ \sigma \qquad \text{iff} \\
\quad \mathsf{S}(m_0, pc_0+1, [0], \mathsf{cl\_}\sigma) \\
\quad \forall y : \forall a : \forall i : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, [i], a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, y, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{getfield}\ f \qquad \text{iff} \\
\quad \forall r : \forall a : \mathsf{S}(m_0, pc_0, [0], r) \wedge \mathsf{H}(r, f, a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, [0], a) \\
\quad \forall y : \forall a : \forall i : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, y, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

$$
\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{getfield this}\ f \qquad \text{iff} \\
\quad \forall r : \forall a : \mathsf{L}(m_0, pc_0, \mathsf{var\_0}, r) \wedge \mathsf{H}(r, f, a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, [0], a) \\
\quad \forall y : \forall a : \forall i : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, [i], a) \ \Rightarrow\ \mathsf{S}(m_0, pc_0+1, y, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \ \Rightarrow\ \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}
$$

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{putfield}\ f$   iff
  $\forall r : \forall a : \mathsf{S}(m_0, pc_0, [1], r) \land \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{H}(r, f, a)$
  $\forall y : \forall a : \forall i : y = [i+2] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)$

$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{putfield this}\ f$   iff
  $\forall r : \forall a : \mathsf{L}(m_0, pc_0, \mathsf{var\_0}, r) \land \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{H}(r, f, a)$
  $\forall y : \forall a : \forall i : y = [i+1] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)$

$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{getstatic}\ f$   iff
  $\forall a : \mathsf{K}(f, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [0], a)$
  $\forall y : \forall a : \forall i : y = [i+1] \land \mathsf{S}(m_0, pc_0, [i], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, y, a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)$

$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{putstatic}\ f$   iff
  $\forall a : \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{K}(f, a)$
  $\forall y : \forall a : \forall i : y = [i+1] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)$
```

Method Support

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{invokestatic}\ m$   iff
  $\forall a : \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(m, 1, \mathsf{var\_0}, a)$
                              $\vdots$
  $\forall a : \mathsf{S}(m_0, pc_0, [|m|-1], a) \Rightarrow \mathsf{L}(m, 1, \mathsf{var\_(|m|-1)}, a)$
  if $m$.returnType $=$ void then
    $\forall y : \forall a : \forall i : y = [i+|m|] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a)$
  if $m$.returnType $\neq$ void then
    $\forall a : \mathsf{S}(m, \mathsf{END}_m, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [0], a)$
    $\forall y : \forall z : \forall a : \forall i : y = [i+|m|] \land z = [i+1] \land$
      $\mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, z, a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)$

$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{invokespecial}\ m$   iff
  $\forall r : \mathsf{S}(m_0, pc_0, [|m|], r) \Rightarrow \mathsf{L}(m, 1, \mathsf{var\_0}, r)$
  $\forall a : \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(m, 1, \mathsf{var\_1}, a)$
                              $\vdots$
  $\forall a : \mathsf{S}(m_0, pc_0, [|m|-1], a) \Rightarrow \mathsf{L}(m, 1, \mathsf{var\_|m|}, a)$
  if $m$.returnType $=$ void then
    $\forall y : \forall a : \forall i : y = [i+|m|+1] \land \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a)$
  if $m$.returnType $\neq$ void then
    $\forall a : \mathsf{S}(m, \mathsf{END}_m, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [0], a)$
    $\forall y : \forall z : \forall a : \forall i : y = [i+|m|+1] \land z = [i+1] \land$
      $\mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, z, a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)$
```

Note that for definite, ie. static and special, methods the end token, END_m, need not be looked up, but can be calculated directly since the invoked method is uniquely and completely determined at compile time unlike for virtual methods.

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{invokevirtual}\ m$   iff
  $\forall r : \forall mv : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \Rightarrow$
    $\mathsf{L}(mv, 1, \mathsf{var\_0}, r)$
  $\forall r : \forall mv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
    $\mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_1}, a)$
                              $\vdots$
  $\forall r : \forall mv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
    $\mathsf{S}(m_0, pc_0, [|m| - 1], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_|m|}, a)$
  if $m.\mathrm{returnType} \neq \texttt{void}$ then
    $\forall y : \forall z : \forall a : \forall i :$
      $y = [i + |m| + 1] \wedge z = [i + 1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, z, a)$
    $\forall r : \forall mv : \forall endmv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
      $\mathsf{END}(mv, endmv) \wedge \mathsf{S}(mv, endmv, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a)$
  if $m.\mathrm{returnType} = \texttt{void}$ then
    $\forall y : \forall a : \forall i : y = [i + |m| + 1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [i], a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$
```

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{invokeinterface}\ m$   iff
  $\forall r : \forall mv : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \Rightarrow$
    $\mathsf{L}(mv, 1, \mathsf{var\_0}, r)$
  $\forall r : \forall mv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
    $\mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_1}, a)$
                              $\vdots$
  $\forall r : \forall mv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
    $\mathsf{S}(m_0, pc_0, [|m| - 1], a) \Rightarrow \mathsf{L}(mv, 1, \mathsf{var\_|m|}, a)$
  if $m.\mathrm{returnType} \neq \texttt{void}$ then
    $\forall a : \forall i : \mathsf{S}(m_0, pc_0, [i + |m| + 1], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, |i + 1|, a)$
    $\forall r : \forall mv : \forall endmv : \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge$
      $\mathsf{END}(mv, endmv) \wedge \mathsf{S}(mv, endmv, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a)$
  if $m.\mathrm{returnType} = \texttt{void}$ then
    $\forall a : \forall i : \mathsf{S}(m_0, pc_0, [i + |m| + 1], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, |i|, a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$
```

When returning from a method, the return value must be copied to the location specified by the end token related to the current method. The end token is computed rather than looked up:

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{return}\ t$   iff
  $\forall a : \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{S}(m_0, \mathsf{END}_{\mathsf{m_0}}, [0], a)$
```

As a special case, `return` without a return value simply generates the always true clause:

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{return}$   iff
  $\mathbf{1}$
```

Array Support.

```
$(\mathsf{K}, \mathsf{H}, \mathsf{L}, \mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{new (array}\ t\texttt{)}$   iff
  $\mathsf{S}(m_0, pc_0 + 1, [0], \mathsf{ar\_t})$
  $\forall y : \forall a : \forall i : y = [i + 1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, y, a)$
  $\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)$
```

$$\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{arraylength} \quad \text{iff} \\
\quad \mathsf{S}(m_0, pc_0+1, [0], \mathsf{INT}) \\
\quad \forall y : \forall a : \forall i : y = [i+1] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, y, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}$$

$$\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{arrayload}\ t \quad \text{iff} \\
\quad \forall r : \forall a : \mathsf{S}(m_0, pc_0, [1], r) \wedge \mathsf{H}(r, \mathsf{ARRAY}, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [0], a) \\
\quad \forall y : \forall z : \forall a : \forall i : \\
\quad\quad y = [i+1] \wedge z = [i+2] \wedge \mathsf{S}(m_0, pc_0, z, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, y, a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}$$

$$\begin{array}{l}
(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{arraystore}\ t \quad \text{iff} \\
\quad \forall r : \forall a : \mathsf{S}(m_0, pc_0, [2], 2) \wedge \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{H}(r, \mathsf{ARRAY}, a) \\
\quad \forall y : \forall a : \forall i : \\
\quad\quad y = [i+3] \wedge \mathsf{S}(m_0, pc_0, y, a) \Rightarrow \mathsf{S}(m_0, pc_0+1, [i], a) \\
\quad \forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a)
\end{array}$$

5 Theoretical Results

In the previous sections we have shown how the Flow Logic specification can be converted into a constraint generating format. In this section we prove that the conversion is correct, in other words: that a solution to the constraints is an acceptable analysis with respect to the Flow Logic specification.

To this end we first define a *concretisation function* on solutions for constraint systems:

Definition 1 *Let* L*,* S*,* K *and* H *be relations defined as in Section 3. We then define*

$$\gamma(\mathsf{L})(m, pc)(x) = \{ v \mid \mathsf{L}(m, pc, x, v) \}$$
$$\gamma(\mathsf{S})(m, pc)|_i = \{ v \mid \mathsf{S}(m, pc, i, v) \}$$
$$\gamma(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) = (\gamma(\mathsf{K}), \gamma(\mathsf{H}), \gamma(\mathsf{L}), \gamma(\mathsf{S}))$$

The above definition formalises the intuitive relation between solutions for constraint systems and analysis estimates. It remains to be shown that this relation does indeed hold; this is stated formally in the following Conjecture:

Fact 2

$$(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{instr} \quad \textit{iff} \quad \gamma((\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S})) \models (m_0, pc_0) : \texttt{instr}$$

Proof (sketch). The fact is proved by case analysis on `instr`.

Case (`push`): We assume that

$$(\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S}) \models_{CG} (m_0, pc_0) : \texttt{push}\ v \qquad (1)$$

and must prove that $\gamma((\mathsf{K},\mathsf{H},\mathsf{L},\mathsf{S})) \models (m_0, pc_0) : \texttt{push}\ v$. From (1) it follows that

$$\begin{array}{lr}
\mathsf{S}(m_0, pc_0+1, [0], \mathsf{INT}) & (2) \\
\forall i : \forall a : \forall y : & \\
\quad (y = [i+1]) \wedge \mathsf{S}(m_0, pc_0, [i], a) \Rightarrow \mathsf{S}(m_0, pc_0+1, y, a) & (3) \\
\forall x : \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0+1, x, a) & (4)
\end{array}$$

From (2) it follows that

$$\gamma(\mathsf{S})(m_0, pc_0 + 1)|_0 \supseteq \{\mathsf{INT}\} \tag{5}$$

and from (3) we have

$$\forall i : \forall a : a \in \gamma(\mathsf{S})(m_0, pc_0)|_i \;\Rightarrow\; v \in \gamma(\mathsf{S})(m_0, pc_0 + 1)|_{i+1}$$

which is equivalent to

$$\forall i : \gamma(\mathsf{S})(m_0, pc_0)|_i \subseteq \gamma(\mathsf{S})(m_0, pc_0 + 1)|_{i+1} \tag{6}$$

Now combining (5) and (6) yields

$$\{\mathsf{INT}\} :: \gamma(\mathsf{S})(m_0, pc_0) \sqsubseteq \gamma(\mathsf{S})(m_0, pc_0 + 1) \tag{7}$$

From (4) we immediately obtain

$$\begin{aligned} &\forall x : \\ &\quad \gamma(\mathsf{L})(m_0, pc_0)(x) \subseteq \gamma(\mathsf{L})(m_0, pc_0 + 1)(x) \end{aligned} \tag{8}$$

which is equivalent to

$$\gamma(\mathsf{L})(m_0, pc_0) \sqsubseteq \gamma(\mathsf{L})(m_0, pc_0 + 1) \tag{9}$$

■

As a result of the intuitive relation, it can be thought of as simply a change of notation, between the Flow Logic specification and the constraint generator the proof is rather straightforward.

6 Optimising the Constraints

In this section we discuss a number of optimisation strategies that can be applied to the constraints defined in the previous sections. Which strategies to employ and whether to employ them at the constraint level or at the specification level will most likely depend on the specific situation; therefore experimentation and benchmarking will be needed to determine what is appropriate for the control flow analysis discuseed in this paper.

6.1 Constraint Tuning

In [1] a number of experiments with the Succinct Solver are performed. These experiments indicate several ways to increase the performance of the solver by careful (re-)formulation of the constraints to be solved. In the following we briefly discuss how the suggestions from [1] can be applied to the constraints defined in this paper.

6.1.1 Parameter Ordering

Earlier versions of the solver were rather sensitive to the particular ordering of parameters in a relation due to the internal representation of relations. Experiments (cf. [1]) show that choosing the right parameter ordering can have a significant impact on the time needed to solve a particular set of constraints.

To overcome this problem the solver now computes all the relevant reorderings of a relation internally, thus allowing it to always choose the best reordering in a given situation. This internal computation is guaranteed to yield the smallest possible number of reorderings needed for binary and tertiary relations. For relations with four or more parameters this cannot be guaranteed.

While the internal computation of all the relevant reorderings can reduce the time needed to solve a set of constraints significantly, it also implies an increase in the space needed to solve the constraints, since a relation may be represented a number of times internally (once for each relevant reordering). For very large constraint sets it may thus be necessary to turn off the internal computation of reorderings at the price of (possibly) increasing the time needed to solve the constraints. In the long term, the problem can be alleviated by implementing more space efficient data structures for the internal representation of the different reorderings.

6.1.2 Conjunct Ordering in Preconditions

Since the solver evaluates preconditions from left to right by propagating variable bindings, called environments, that satisfy subclauses in the precondition, time savings can be achieved by moving subclauses that only allow a small number of environments, ie. subclauses that are only true for a small number of values, more to the left in a precondition, since this reduces the number of environments passed further on in the evaluation of a precondition.

This can be illustrated by looking at the constraints for the `invokevirtual` instruction. Here the constraints for copying the arguments forward to the invoked method are of the following form:

$$\begin{array}{l} \forall r : \forall mv : \forall a : \\ \quad \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \\ \qquad \mathsf{L}(mv, 1, \mathsf{var\_1}, a) \end{array}$$

Since the ML relation is expected to be quite sparse, the above clause should be rewritten to exploit this fact, ie.:

$$\begin{array}{l} \forall r : \forall mv : \forall a : \\ \quad \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \\ \qquad \mathsf{L}(mv, 1, \mathsf{var\_1}, a) \end{array}$$

Similarly for the END-tokens (still for the `invokevirtual` instruction):

$$\begin{array}{l} \forall r : \forall mv : \forall endmv : \forall a : \\ \quad \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \mathsf{END}(mv, endmv) \wedge \\ \qquad \mathsf{S}(mv, endmv, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a) \end{array}$$

Here both the ML and the END relations are expected to be quite sparse, thus the above should be rewritten as

$$\begin{array}{l}\forall r : \forall mv : \forall endmv : \forall a : \\ \quad \mathsf{END}(mv, endmv) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \\ \qquad \mathsf{S}(mv, endmv, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a)\end{array}$$

Such optimisations are possible for several instructions.

6.1.3 Existential Quantification in Preconditions

Another technique for optimising the time requirement of a set of constraints is to change universal quantifiers into existential quantifiers where possible without changing the meaning of a constraint. In general this is possible in the case where a universally quantified variable is used in the precondition but not in the conclusion of an implication, ie. let x be a variable that is *not* free in pre_y and *con* then a clause of the following form

$$\forall x : \forall y_1 : \cdots \forall y_n : (pre_x \wedge pre_y) \Rightarrow conc$$

should be converted into

$$\forall y_1 : \cdots \forall y_n : ((\exists x : pre_x) \wedge pre_y) \Rightarrow conc$$

This optimisation is illustrated using the `invokevirtual` instruction, where the clauses that model the parameter transfer are of the form:

$$\begin{array}{l}\forall r : \forall mv : \forall a : \\ \quad \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \\ \qquad \mathsf{L}(mv, 1, \mathsf{var\_1}, a)\end{array}$$

Applying the above optimisation rule, such a constraint can be rewritten to the following:

$$\begin{array}{l}\forall mv : \forall a : \\ \quad (\exists r : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv)) \wedge \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \\ \qquad \mathsf{L}(mv, 1, \mathsf{var\_1}, a)\end{array}$$

6.1.4 Sharing Preconditions

It is possible to improve the time and space needed (by a constant factor) by merging clauses with the same preconditions but different conclusions. This then avoids recomputing the precondition for each different conclusion. In symbols, clauses on the form

$$(pre \Rightarrow conc_1) \wedge (pre \Rightarrow conc_2)$$

are converted into clauses where the shared precondition is made explicit:

$$pre \Rightarrow (conc_1 \wedge conc_2)$$

We illustrate this conversion by applying it to the `invokevirtual` instruction:

$$
\begin{array}{l}
\forall r \forall mv : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \Rightarrow \\
\quad \mathsf{L}(mv, 1, [0], r) \\
\forall r \forall mv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \\
\quad \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(mv, 1, [1], a) \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\forall r \forall mv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \\
\quad \mathsf{S}(m_0, pc_0, [|m|-1], a) \Rightarrow \mathsf{L}(mv, 1, [|m|], a) \\
\forall a \forall i : \mathsf{S}(m_0, pc_0, [i + |m| + 1], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, |i+1|, a) \\
\forall r \forall mv \forall endmv \forall a : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \wedge \\
\quad \mathsf{END}(mv, endmv) \wedge \mathsf{S}(mv, endmv, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a) \\
\forall x \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)
\end{array}
$$

is converted into

$$
\begin{array}{l}
\forall r \forall mv : \mathsf{S}(m_0, pc_0, [|m|], r) \wedge \mathsf{ML}(m.\mathrm{id}, r, mv) \Rightarrow \\
\quad \mathsf{L}(mv, 1, [0], r) \\
\quad \forall a : \\
\qquad \mathsf{S}(m_0, pc_0, [0], a) \Rightarrow \mathsf{L}(mv, 1, [1], a) \\
\qquad \vdots \\
\qquad \mathsf{S}(m_0, pc_0, [|m|-1], a) \Rightarrow \mathsf{L}(mv, 1, [|m|], a) \\
\qquad \forall endmv : \\
\qquad\quad \mathsf{END}(mv, endmv) \wedge \mathsf{S}(mv, endmv, [0], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [0], a) \\
\forall a : \forall i : \mathsf{S}(m_0, pc_0, [i + |m| + 1], a) \Rightarrow \mathsf{S}(m_0, pc_0 + 1, [i+1], a) \\
\forall x \forall a : \mathsf{L}(m_0, pc_0, x, a) \Rightarrow \mathsf{L}(m_0, pc_0 + 1, x, a)
\end{array}
$$

6.2 Path Compression

Rather than “tweaking” the individual clauses, a more global approach could be taken, ie. looking at all the generated clauses. With this approach we can identify paths where information is just copied forward and compress them. For example:

$$\hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + n_1) \sqsubseteq \cdots \sqsubseteq \hat{S}(m_0, pc_0 + n_i)$$

can be compressed to

$$\hat{S}(m_0, pc_0) \sqsubseteq \hat{S}(m_0, pc_0 + n_i)$$

and thus eliminating all the intermediate steps. This can be done *if and only if* all the intermediate variables, ie. $\hat{S}(m_0, pc_0 + n_2), \ldots, \hat{S}(m_0, pc_0 + n_{i-1})$, have exactly one definition and exactly one use.

The clauses that arise from the control flow analysis are especially likely to have such paths for local variables; most instructions do not access or modify local variables, but the analysis copies them forward nevertheless. Instead of finding such paths in the clauses, it would also be possible to lift path compression directly into the flow logic specification of the control flow anlaysis by only copying information forward to where it is needed.

6.3 Better Representation of the Abstract Domains

Another approach to reducing space required by the constraints is to find better, ie. less space consuming, representations of the abstract domains. The stack and the local heap are obvious candidates for such a representation change, especially in combination with path compression as discussed in Section 6.2.

For the stack a possible alternative to the current representation is to use *stack variables* rather than the full (abstract) stack. The idea is to use placeholders to keep track of the possible stack layouts and then for each placeholder keeping track of the possible values that placeholder can have. The advantage is that it eliminates the need for copying the full set of possible values for a stack position, instead only the placeholder is copied. We are currently working on ideas along these lines, in particular with an eye to using the optimised representation when implementing the data flow analysis discussed in [3].

7 Prototype Implementation

In order to obtain feedback on the analysis and constraint generation as early in the process as possible, a simple constraint generation prototype has been implemented in Haskell (cf. `www.haskell.org`). We shall not go into any implementation details here, but the prototype is available from the author upon request.

The prototype consists of a simple parser, that parses af simplified version of Carmel (called CarmelLight), and a constraint generator for the Succinct Solver. The choice of parsing a simplified Carmel rather than "real" Carmel syntax was made to cut down on the development time of the prototype.

Because of the lack of benchmark programs available in the Carmel syntax, as defined in [6], and the absence of an automatic translator from JavaCard to Carmel, a number of simplifying assumptions were made to facilitate the (manual) translation from JavaCard to Carmel. In particular: dynamic method lookup (via the *methodLookup* function) was replaced by a simpler notion by assuming that inherited methods are (syntactically) copied to the relevant classes. Furthermore, instead of using the address scheme suggested in [6], based on method and class *identities*, a simpler addressing scheme is used, based on class and method *names* instead. The simpler addressing scheme is sufficient when the source code of the entire program being analysed is available.

The above simplifying assumptions only change or influence the analysis in minor ways, but they do make it easier and faster to (manually) convert or construct benchmark programs that can be used with the prototype implementation of the analysis.

In Section 7.2 consequences of the above choices are discussed along with suggestions for improvements.

7.1 Using the Prototype

In Listing 1 we give an example of a small CarmelLight program. Note that the `#direct{...}` command used at the bottom of Listing 1 is used to send clauses directly to the solver, ie. with no interference from the constraint generator. In this example it is used to encode the maximum stack height; this also ensures that the universe of the solver contains encodings of all the stack

```
program Test {
  class sigma1
  {
    method m1 1
    {
      push 42
      return
    }

    method m2 1
    {
      new sigma1
      push 1
      invokevirtual (sigma1,m1,1)
      store 1
      return
    }
  }
}

#direct{
U(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(zero))))))))))
}
```

Listing 1: Example program in CarmelLight

positions of relevance. Due to the way the solver is implemented and becuase the stack positions are encoded as nested terms, only the maximum number needs to be specified, the remaining numbers, ie. all *subterms* of the encoding, are automatically constructed by the solver.

Listing 2 shows an excerpt of the constraints generated by the prototype implementation. The excerpt shows the clauses generated for a method invocation. The full set of constraints generated is shown in Appendix A.1.

Listing 3 illustrates the output from the solver. The excerpt illustrates the solution for the stack and for the local heap. The full solution is shown in Appendix A.2.

7.2 Testing the Prototype

The prototype has been tested on a number of small ad-hoc test programs as well as two Sun JavaCard demo applets: Wallet and JavaPurse (translated from JavaCard to CarmelLight by hand).

The Wallet applet consists of approximately 200 lines of CarmelLight code while the JavaPurse has around 1300 lines of CarmelLight code. Both applets make use of various JavaCard API's; a simplified model of the relevant API calls (taking only rudimentary control flow into account) were implemented in order to carry out the tests.

Solving the constraints generated for the two test programs took approximately 4 seconds and 22 seconds for the Wallet and JavaPurse respectively. An

```
/* ("sigma1","m2",3): invokevirtual (sigma1,m1) */
(A r. S(cl_sigma1,m2,pc_3,suc(zero),r) => L(r,m1,pc_1,var_0,r)) &
(A r. A a. S(cl_sigma1,m2,pc_3,suc(zero),r) &
 S(cl_sigma1,m2,pc_3,zero,a) => L(r,m1,pc_1,var_1,a)) &
1 &
(A r. A end. A a. S(cl_sigma1,m2,pc_3,suc(zero),r) &
 End(r,m1,end) & S(r,m1,end,zero,a) =>
   S(cl_sigma1,m2,pc_4,zero,a)) &
(A x. A y. A i. A a. x = suc(suc(i)) & y = suc(i) &
 S(cl_sigma1,m2,pc_3,x,a) => S(cl_sigma1,m2,pc_4,y,a)) &
(A x. A a. L(cl_sigma1,m2,pc_3,x,a) => L(cl_sigma1,m2,pc_4,x,a)) &
```

Listing 2: Excerpt of constraints generated for example program

```
Relation S/5:
  (cl_sigma1, m2, pc_4, zero, INT),
  (cl_sigma1, m2, pc_3, suc (zero), cl_sigma1),
  (cl_sigma1, m2, pc_3, zero, INT),
  (cl_sigma1, m2, pc_2, zero, cl_sigma1),
  (cl_sigma1, m1, end_sigma1_m1, zero, INT),
  (cl_sigma1, m1, pc_2, zero, INT),

Relation L/5:
  (cl_sigma1, m2, pc_5, var_1, INT), (cl_sigma1, m1, pc_2, var_1, INT),
  (cl_sigma1, m1, pc_2, var_0, cl_sigma1),
  (cl_sigma1, m1, pc_1, var_1, INT),
  (cl_sigma1, m1, pc_1, var_0, cl_sigma1),
```

Listing 3: Excerpt of solver output for example program

estimate[1] of the memory used by the solver for the Wallet program is approximately 15 MB and approximately 30 MB for the JavaPurse program.

While the above few tests do not constitute a formal and rigorous benchmark test, they do seem to indicate that the analysis will be fast enough to be useful. Memory consumption on the other hand should be watched carefully and methods for lowering the space requirements of the analysis, eg. those discussed in Section 6, should be investigated and possibly implemented if the analysis is to be used on realistic sized programs.

The current prototype, while helpful for giving a first impression of the analysis' precision and performance, is not adequate for performing a systematic benchmark test of the analysis and the various optimisation strategies discussed in Section 6. The scarcity of "real world" examples and the work required to manually translate these into CarmelLight is prohibitive for such a study. Furthermore, for such experiments to be useful it would be necessary to implement better and more complete models of the API's and the JavaCard Runtime Environment (JCRE) in the semantics and the analysis. Currently there is work under way at Imperial College (London) to model parts of the API's and the JCRE in the operational semantics, cf. [9]. Work has also been done, by Luke Jackson at Imperial College, on implementing a Carmel interpreter, including of course a parser for the full Carmel language, cf. [5]. This opens for the possibility of implementing the constraint generation described in this paper on top of that system, thus overcoming at least the problems created by basing the prototype on CarmelLight. It could also pave the way for a systematic benchmark testing of both constraint generation and optimisation strategies.

8 Conclusion and Future Work

In this paper we have shown how the Flow Logic specification of a control flow analysis for Carmel, defined in [4], can be converted to a constraint generator format over the Alternation-free Least Fixed-Point (ALFP) logic. As a part of this conversion we have discussed how the abstract domains used by the Flow Logic specification can be represented in ALFP and we have sketched a proof of semantic soundness for the conversion. Strategies for improving the generated constraints with respect to time and/or space needed to solve them were briefly reviewed and a prototype implementation, generating constraints for the Succinct Solver, was discussed.

In the near future the extensions of the control flow analysis specified in [3] will be converted to a similar constraint generator format. This work is already under way, focusing at first on the ownership and exception analyses; in [2] a brief overview of the implementation of ownership analysis is given and a way of using it to verify that Carmel programs do not violate the JCRE firewall is discussed. Proving the semantic correctness of the entire conversion is also a priority in the short term.

In the longer term, a better prototype should be implemented. This should include a better parser (for the full Carmel language), good support for extensions and more ways of manipulating and displaying the analysis result. Most, if not all, of these features are already available in the Carmel Interpreter by

[1]Based on program allocation during execution, as reported by the Unix commands `top` and `ps`.

Luke Jackson [5]. It would be a very straightforward task to implement the constraint generation described in this document on top of the system described in [5]. For these reasons we believe that the mentioned system would form an ideal basis for a vastly improved prototype implementation of the constraint generation better suited for handling programs of realistic size.

References

[1] Mikael Buchholtz, Hanne Riis Nielson, and Flemming Nielson. Experiments with Succinct Solvers. SECSAFE-IMM-002-1.0. Also published as DTU Technical Report IMM-TR-2002-4, February 2002.

[2] René Rydhof Hansen. A prototype tool for JavaCard firewall analysis. In *Nordic Workshop on Secure IT-Systems, NordSec'02*, Karlstad, Sweden, November 2002. To appear.

[3] René Rydhof Hansen. Extending the Flow Logic for Carmel. SECSAFE-IMM-003-1.0, 2002.

[4] René Rydhof Hansen. Flow Logic for Carmel. SECSAFE-IMM-001-1.5, 2002.

[5] Luke Jackson. Carmel interpreter. Web page, 2002. URL: `http://www.doc.ic.ac.uk/~siveroni/secsafe/docs/interpreter/`.

[6] Renaud Marlet. Syntax of the JCVM Language To Be Studied in the SecSafe Project. SECSAFE-TL-005-1.7, May 2001.

[7] Flemming Nielson and Helmut Seidl. Control-Flow Analysis in Cubic Time. In *Proc. ESOP'01*, April 2001. SECSAFE-DAIMI-006-1.0 (preprint).

[8] Flemming Nielson and Helmut Seidl. Succinct solvers. Technical Report 01-12, University of Trier, Germany, 2001.

[9] Igor Siveroni and Chris Hankin. A Proposal for the JCVMLe Operational Semantics. SECSAFE-ICSTM-001-2.2, October 2001.

A Example

A.1 Generated Constraints

```
/* Carmel Constraint Generator [version 1.14-alpha] */
/* Constraints for program: Test */
/* Class: sigma1 */
/* Method: (sigma1,m1) */

/* ("sigma1","m1",1): push 42 */
S(cl_sigma1,m1,pc_2,zero,INT) &
(A y. A i. A a. y = suc(i) & S(cl_sigma1,m1,pc_1,i,a) =>
  S(cl_sigma1,m1,pc_2,y,a)) &
(A x. A a. L(cl_sigma1,m1,pc_1,x,a) => L(cl_sigma1,m1,pc_2,x,a)) &
```

```
/* ("sigma1","m1",2): return */
(A a. S(cl_sigma1,m1,pc_2,zero,a) =>
  S(cl_sigma1,m1,end_sigma1_m1,zero,a)) &
End(cl_sigma1,m1,end_sigma1_m1) &
/* Method: (sigma1,m2) */

/* ("sigma1","m2",1): new sigma1 */
S(cl_sigma1,m2,pc_2,zero,cl_sigma1) &
(A y. A i. A a. y = suc(i) & S(cl_sigma1,m2,pc_1,i,a) =>
  S(cl_sigma1,m2,pc_2,y,a)) &
(A x. A a. L(cl_sigma1,m2,pc_1,x,a) => L(cl_sigma1,m2,pc_2,x,a)) &

/* ("sigma1","m2",2): push 1 */
S(cl_sigma1,m2,pc_3,zero,INT) &
(A y. A i. A a. y = suc(i) & S(cl_sigma1,m2,pc_2,i,a) =>
  S(cl_sigma1,m2,pc_3,y,a)) &
(A x. A a. L(cl_sigma1,m2,pc_2,x,a) => L(cl_sigma1,m2,pc_3,x,a)) &

/* ("sigma1","m2",3): invokevirtual (sigma1,m1) */
(A r. S(cl_sigma1,m2,pc_3,suc(zero),r) => L(r,m1,pc_1,var_0,r)) &
(A r. A a. S(cl_sigma1,m2,pc_3,suc(zero),r) &
  S(cl_sigma1,m2,pc_3,zero,a) => L(r,m1,pc_1,var_1,a)) &
1 &
(A r. A end. A a. S(cl_sigma1,m2,pc_3,suc(zero),r) &
  End(r,m1,end) & S(r,m1,end,zero,a) =>
    S(cl_sigma1,m2,pc_4,zero,a)) &
(A x. A y. A i. A a. x = suc(suc(i)) & y = suc(i) &
  S(cl_sigma1,m2,pc_3,x,a) => S(cl_sigma1,m2,pc_4,y,a)) &
(A x. A a. L(cl_sigma1,m2,pc_3,x,a) => L(cl_sigma1,m2,pc_4,x,a)) &

/* ("sigma1","m2",4): store 1 */
(A a. S(cl_sigma1,m2,pc_4,zero,a) => L(cl_sigma1,m2,pc_5,var_1,a)) &
(A x. A i. A a. x = suc(i) & S(cl_sigma1,m2,pc_4,x,a) =>
  S(cl_sigma1,m2,pc_5,i,a)) &
(A x. A a. x != var_1 & L(cl_sigma1,m2,pc_4,x,a) =>
  L(cl_sigma1,m2,pc_5,x,a)) &

/* ("sigma1","m2",5): return */
(A a. S(cl_sigma1,m2,pc_5,zero,a) =>
  S(cl_sigma1,m2,end_sigma1_m2,zero,a)) &
End(cl_sigma1,m2,end_sigma1_m2)
 &
/* Special Constraints */
U(suc(suc(suc(suc(suc(suc(suc(suc(suc(suc(zero)))))))))))
```

Listing 4: Constraints generated for example program

A.2 Solver Output

```
The Universe:   (cl_sigma1, m1, pc_2, zero, INT, pc_1, end_sigma1_m1,
  m2, pc_3, suc (zero), var_0, var_1, pc_4, pc_5, end_sigma1_m2,
  suc (suc (suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))))),
  suc (suc (suc (suc (suc (suc (suc (suc (suc (zero))))))))),
  suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))),
  suc (suc (suc (suc (suc (suc (suc (zero))))))),
  suc (suc (suc (suc (suc (suc (zero)))))),
  suc (suc (suc (suc (suc (zero))))),
  suc (suc (suc (suc (zero)))),
  suc (suc (suc (zero))), suc (suc (zero)))

Relation S/5:
  (cl_sigma1, m2, pc_4, zero, INT),
  (cl_sigma1, m2, pc_3, suc (zero), cl_sigma1),
  (cl_sigma1, m2, pc_3, zero, INT),
  (cl_sigma1, m2, pc_2, zero, cl_sigma1),
  (cl_sigma1, m1, end_sigma1_m1, zero, INT),
  (cl_sigma1, m1, pc_2, zero, INT),

Relation suc/2:
  (suc (zero), zero), (suc (suc (zero)), suc (zero)),
  (suc (suc (suc (zero))), suc (suc (zero))),
  (suc (suc (suc (suc (zero)))), suc (suc (suc (zero)))),
  (suc (suc (suc (suc (suc (zero))))), suc (suc (suc (suc (zero))))),
  (suc (suc (suc (suc (suc (suc (zero)))))), suc (suc (suc (suc (suc
    (zero)))))),
  (suc (suc (suc (suc (suc (suc (suc (zero))))))), suc (suc (suc (suc
    (suc (suc (zero))))))),
  (suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))), suc (suc (suc
    (suc (suc (suc (suc (zero)))))))),
  (suc (suc (suc (suc (suc (suc (suc (suc (suc (zero))))))))), suc
    (suc (suc (suc (suc (suc (suc (suc (zero))))))))),
  (suc (suc (suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))))),
    suc (suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))))),

Relation L/5:
  (cl_sigma1, m2, pc_5, var_1, INT), (cl_sigma1, m1, pc_2, var_1, INT),
  (cl_sigma1, m1, pc_2, var_0, cl_sigma1),
  (cl_sigma1, m1, pc_1, var_1, INT),
  (cl_sigma1, m1, pc_1, var_0, cl_sigma1),

Relation End/3:
  (cl_sigma1, m2, end_sigma1_m2), (cl_sigma1, m1, end_sigma1_m1),

Relation U/1:
  (suc (suc (suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))))),

Relation suc-/2:
  (suc (suc (suc (suc (suc (suc (suc (suc (suc (zero))))))))), suc
    (suc (suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))))),
  (suc (suc (suc (suc (suc (suc (suc (suc (zero)))))))), suc (suc (suc
    (suc (suc (suc (suc (suc (suc (zero))))))))),
  (suc (suc (suc (suc (suc (suc (suc (zero))))))), suc (suc (suc (suc
    (suc (suc (suc (suc (zero)))))))),
  (suc (suc (suc (suc (suc (suc (zero)))))), suc (suc (suc (suc (suc
    (suc (suc (zero))))))),
  (suc (suc (suc (suc (suc (zero))))), suc (suc (suc (suc (suc (suc
    (zero)))))),
  (suc (suc (suc (suc (zero)))), suc (suc (suc (suc (suc (zero))))),
  (suc (suc (suc (zero))), suc (suc (suc (suc (zero))))),
  (suc (suc (zero)), suc (suc (suc (zero)))),
  (suc (zero), suc (suc (zero))),
  (zero, suc (zero)),
```

Listing 5: Result from solver for example program