Linguagem de Especificação Leve Hoare-Separação para Java

Tiago Vieira Correia dos Santos
Orientador: Luís Caires

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

18 de Novembro de 2010

Objectivos

O trabalho realizado teve como principais objectivos:

- Estudar técnicas para melhorar a usabilidade da verificação estática em programas "reais";
- O desenvolvimento de uma linguagem lógica de especificação leve para Java;
- Esta linguagem tem como base a lógica de Hoare, por um lado simplificada por restrição a um fragmento proposicional, por outro enriquecida com tratamento de "aliasing";
- A extensão da implementação de um compilador do Java para suportar tais especificações leves, complementando a análise de tipos normalmente já efectuada;
- Chamamos SPECJava ao nosso sistema / compilador.

Exemplo em SpecJava

```
public class Entry {
  private int x;
  invariant x:pos;
  private Entry next;

  public Entry(int x, Entry n)
    requires x:pos
  {
    this.x = x;
    this.next = n;
  }

  public Entry getNext()
  { return next; }

  public int getElement()
    ensures return:pos
  { return x; }
}
```

Verificação Formal de Programas

É uma área de investigação há mais de 40 anos que tem assistido a um novo impulso recentemente.

Técnicas de Verificação agrupadas em:

- Verificação Estática
  - Sistemas de Tipos – detecção de erros de tipificação no programa (e.g. uso de instruções ilegais numa linguagem)
  - Baseada em Lógicas – verificação da correcção do programa face às suas especificações.
- Verificação Dinâmica
  - Cobertura de Código – testes de input;
  - Model Checking – pesquisa exaustiva de estados.

Neste trabalho focamo-nos na verificação estática de programas.

Verificação Estática

Actualmente os sistemas de tipos desempenham um papel fulcral numa linguagem de programação:

- permitem detectar alguns tipos de erros em tempo de compilação;
- são decidíveis;
- Em geral não abrangem tipos de erros mais complexos, como por exemplo:
  - quebra de protocolos;
  - desreferenciações nulas;
  - quebra de integridade (expressa por pré e pós-condições, invariantes).

Quanto às verificações lógicas formais estas:

- tendem a ser indecidíveis;
- não estão totalmente integradas nas linguagens de programação.

Lógicas de Verificação – Técnicas base

A lógica de Hoare [Hoa69] permite raciocinar sobre a correcção de programas, através de um sistema formal onde as especificações são expressas sob a forma de triplos de Hoare ($\{Q\}\ ST\ \{R\}$).

- difícil mecanizar provas – quando aplicar a regra da dedução?

A técnica do cálculo de pré-condições mais fracas proposta por Dijkstra [Dij75], permite verificar a validade de um triplo de Hoare, atribuindo um predicado transformador a cada instrução da linguagem de programação.

- fornece um algoritmo eficaz que reduz o problema da verificação de um triplo de Hoare a um de provas de fórmulas lógicas;
- tal como a lógica de Hoare, não lida com situações de *aliasing*.

Lógicas de Verificação

Lógicas de Verificação:

- Lógica de Hoare
  - Flo67, Hoa69, Lam80
- Cálculo de Pré-condições mais Fracas
  - Dij75, Rey02
- Lógica de Separação
  - Rey02, OHRY01, PB05

Linguagens de Especificação:

- JML (*Java Modeling Language*)
  - LBR99
- OCL (*Object Constraint Language*)

Ferramentas para a Verificação de Programas

Existem várias ferramentas e linguagens de programação que fornecem suporte à verificação de programas:

- Spec#
- ESC/Java2
- KRAKATOA
- JACK
- jStar
- Forge

- auxiliam o programador no processo de desenvolvimento de um programa;
- detectam erros em tempo de compilação e/ou em runtime;
- as ferramentas encontram-se separadas do processo de compilação da linguagem de programação.
- as linguagens de especificação usadas são muito expressivas, mas complexas.
- algumas destas ferramentas não satisfazem as propriedades de completude e/ou correcção.

SpecJava

Ideias base do SPECJava

- Linguagem de especificação inspirada na JML, MAS:
  - leve, porque é baseada numa lógica proposicional monádica
  - sem quantificadores, para garantir decidibilidade
- Verificação baseada em cálculo de pré-condições mais fracas:
  - directamente a partir das construções sintácticas do Java
  - condições verificadas recorrendo a um SMT-Solver
  - efectuada de forma automática em tempo de compilação
- Controlo de *aliasing* por separação de propriedade puras e lineares.
- Esta separação é efectuada numa lógica dual (também contribuição desta tese) que divide as fórmulas em dois contextos (uma parte pura, e outra linear).

Porquê a Lógica Dual Hoare-Separação ?

Motivação

Seja $\{Q\}\ ST\ \{R\}$ um triplo de Hoare, onde Q e R são pré e pós-condições e ST corresponde a um sequência de comandos.

Considerando o exemplo seguinte:

$$\begin{array}{c}\{\neg Open(f) \wedge \neg Open(g)\}\\ f.open();\\ g.open();\\ g.close();\\ f.close();\\ \{true\}\end{array}$$

$$\frac{x : File}{\{\neg Open(x)\}\ x.open()\ \{Open(x)\}}$$

$$\frac{x : File}{\{Open(x)\}\ x.close()\ \{\neg Open(x)\}}$$

Como distinguir **f** e **g** quando são objectos diferentes? E quando são o mesmo objecto (*aliasing*)?

Para lidar com o *aliasing* no contexto de uma lógica leve, definimos uma nova abordagem, combinando a lógica de Hoare com a lógica de Separação, de uma forma bem estruturada.

Lógica Dual Hoare-Separação

Um triplo de Hoare em Lógica Dual Hoare-Separação escreve-se na forma:

$$\{Q + S\}\ ST\ \{R + T\}$$

Onde $Q + S$ e $R + T$, são fórmulas duais, respectivamente pré e pós-condições e ST corresponde a uma sequência de comandos

ψ	$::=$	$\phi + \varphi$	(Fórmula Dual)
φ	$::=$	$\emptyset \mid \phi \mid \phi * \varphi$	(Fórmula Linear)
ϕ	$::=$		(Fórmula Clássica)
		$\bot$	(Absurdo)
	$\mid$	$\phi\ lc\ \phi$	(Fórmula Binária)
	$\mid$	$\neg\phi$	(Negação)
	$\mid$	(ϕ)	(Fórmula entre Parênteses)
	$\mid$	$P(t_1, t_2, \cdots, t_n)$	(Símbolos de Predicado)
lc	$::=$	$\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow$	(Conectivos Lógicos)
t	$::=$		(Termos)
		c	(Constantes)
	$\mid$	x	(Variáveis)
	$\mid$	$f(t_1, t_2, \cdots, t_n)$	(Símbolos de Função)

Lógica Dual Hoare-Separação

Um triplo de Hoare em Lógica Dual Hoare-Separação escreve-se na forma:

$$\{Q + S\}\ ST\ \{R + T\}$$

Onde $Q + S$ e $R + T$, são fórmulas duais, respectivamente pré e pós-condições e ST corresponde a uma sequência de comandos

ψ	$::=$	$\phi + \varphi$	(Fórmula Dual)
φ	$::=$	$\emptyset \mid \phi \mid \phi * \varphi$	(Fórmula Linear)
ϕ	$::=$		(Fórmula Clássica)
		$\bot$	(Absurdo)
	$\mid$	$\phi\ lc\ \phi$	(Fórmula Binária)
	$\mid$	$\neg\phi$	(Negação)
	$\mid$	(ϕ)	(Fórmula entre Parênteses)
	$\mid$	$P(t_1, t_2, \cdots, t_n)$	(Símbolos de Predicado)
lc	$::=$	$\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow$	(Conectivos Lógicos)
t	$::=$		(Termos)
		c	(Constantes)
	$\mid$	x	(Variáveis)
	$\mid$	$f(t_1, t_2, \cdots, t_n)$	(Símbolos de Função)

Lógica Dual Hoare-Separação

Um triplo de Hoare em Lógica Dual Hoare-Separação escreve-se na forma:

$$\{Q + S\}\ ST\ \{R + T\}$$

Onde $Q + S$ e $R + T$, são fórmulas duais, respectivamente pré e pós-condições e ST corresponde a uma sequência de comandos

ψ	$::=$	$\phi + \varphi$	(Fórmula Dual)
φ	$::=$	$\emptyset \mid \phi \mid \phi * \varphi$	(Fórmula Linear)
ϕ	$::=$		(Fórmula Clássica)
		$\bot$	(Absurdo)
	$\mid$	$\phi\ lc\ \phi$	(Fórmula Binária)
	$\mid$	$\neg\phi$	(Negação)
	$\mid$	(ϕ)	(Fórmula entre Parênteses)
	$\mid$	$P(t_1, t_2, \cdots, t_n)$	(Símbolos de Predicado)
lc	$::=$	$\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow$	(Conectivos Lógicos)
t	$::=$		(Termos)
		c	(Constantes)
	$\mid$	x	(Variáveis)
	$\mid$	$f(t_1, t_2, \cdots, t_n)$	(Símbolos de Função)

Programas

program	::=	*classDecl**	(Programa)
classDecl	::=	**class** *cn* { *classMember** }	(Declaração de Classe)
classMember	::=	...	(Membros de Classe)
	|	*field*	(Decl. de Variáveis de Instância)
	|	*method*	(Declaração de Métodos)
	|	*constructor*	(Declaração de Construtores)
	|	*classSpec*	(Especificação de Classe)
classSpec	::=		(Especificação de Classe)
		define *sn*;	(Definição Abstracta)
	|	**define** *sn* = *D*;	(Definição Concreta)
	|	**invariant** *D*;	(Invariante de Classe)
field	::=	*T fn* [= *E*]? ;	(Decl. de Variáveis de Instância)
method	::=	*modifier T mn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Métodos)
constructor	::=	*modifier cn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Construtores)
modifier	::=	**public** | **static** | ... | **pure**	(Modificadores)
spec	::=		(Especificação de Procedimentos)
		requires *D*	(Pré-condição)
	|	**ensures** *D*	(Pós-condição)
ST	::=	...	(Comandos)
	|	**assume** *D*	(Assume)
	|	**sassert** *D*	(Assert Estático)

cn $\in$ nomes das classes *sn* $\in$ nomes dos estados

fn $\in$ nomes das variáveis de instância

Programas

program	::=	*classDecl**	(Programa)
classDecl	::=	**class** *cn* { *classMember** }	(Declaração de Classe)
classMember	::=	...	(Membros de Classe)
	|	*field*	(Decl. de Variáveis de Instância)
	|	*method*	(Declaração de Métodos)
	|	*constructor*	(Declaração de Construtores)
	|	*classSpec*	(Especificação de Classe)
classSpec	::=		(Especificação de Classe)
		define *sn*;	(Definição Abstracta)
	|	**define** *sn* = *D*;	(Definição Concreta)
	|	**invariant** *D*;	(Invariante de Classe)
field	::=	*T fn* [= *E*]? ;	(Decl. de Variáveis de Instância)
method	::=	*modifier T mn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Métodos)
constructor	::=	*modifier cn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Construtores)
modifier	::=	**public** | **static** | ... | **pure**	(Modificadores)
spec	::=		(Especificação de Procedimentos)
		requires *D*	(Pré-condição)
	|	**ensures** *D*	(Pós-condição)
ST	::=	...	(Comandos)
	|	**assume** *D*	(Assume)
	|	**sassert** *D*	(Assert Estático)

cn $\in$ nomes das classes *sn* $\in$ nomes dos estados

fn $\in$ nomes das variáveis de instância

Programas

program	::=	*classDecl**	(Programa)
classDecl	::=	**class** *cn* { *classMember** }	(Declaração de Classe)
classMember	::=	...	(Membros de Classe)
	|	*field*	(Decl. de Variáveis de Instância)
	|	*method*	(Declaração de Métodos)
	|	*constructor*	(Declaração de Construtores)
	|	*classSpec*	(Especificação de Classe)
classSpec	::=		(Especificação de Classe)
		define *sn*;	(Definição Abstracta)
	|	**define** *sn* = *D*;	(Definição Concreta)
	|	**invariant** *D*;	(Invariante de Classe)
field	::=	*T fn* [= *E*]? ;	(Decl. de Variáveis de Instância)
method	::=	*modifier T mn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Métodos)
constructor	::=	*modifier cn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Construtores)
modifier	::=	**public** | **static** | ... | **pure**	(Modificadores)
spec	::=		(Especificação de Procedimentos)
		requires *D*	(Pré-condição)
	|	**ensures** *D*	(Pós-condição)
ST	::=	...	(Comandos)
	|	**assume** *D*	(Assume)
	|	**sassert** *D*	(Assert Estático)

cn $\in$ nomes das classes *sn* $\in$ nomes dos estados

fn $\in$ nomes das variáveis de instância

Programas

program	::=	*classDecl**	(Programa)
classDecl	::=	**class** *cn* { *classMember** }	(Declaração de Classe)
classMember	::=	...	(Membros de Classe)
	|	*field*	(Decl. de Variáveis de Instância)
	|	*method*	(Declaração de Métodos)
	|	*constructor*	(Declaração de Construtores)
	|	*classSpec*	(Especificação de Classe)
classSpec	::=		(Especificação de Classe)
		define *sn*;	(Definição Abstracta)
	|	**define** *sn* = *D*;	(Definição Concreta)
	|	**invariant** *D*;	(Invariante de Classe)
field	::=	*T fn* [= *E*]? ;	(Decl. de Variáveis de Instância)
method	::=	*modifier T mn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Métodos)
constructor	::=	*modifier cn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Construtores)
modifier	::=	**public** | **static** | ... | **pure**	(Modificadores)
spec	::=		(Especificação de Procedimentos)
		requires *D*	(Pré-condição)
	|	**ensures** *D*	(Pós-condição)
ST	::=	...	(Comandos)
	|	**assume** *D*	(Assume)
	|	**sassert** *D*	(Assert Estático)

cn $\in$ nomes das classes *sn* $\in$ nomes dos estados

fn $\in$ nomes das variáveis de instância

Programas

program	::=	*classDecl**	(Programa)
classDecl	::=	**class** *cn* { *classMember** }	(Declaração de Classe)
classMember	::=	...	(Membros de Classe)
	|	*field*	(Decl. de Variáveis de Instância)
	|	*method*	(Declaração de Métodos)
	|	*constructor*	(Declaração de Construtores)
	|	*classSpec*	(Especificação de Classe)
classSpec	::=		(Especificação de Classe)
		define *sn*;	(Definição Abstracta)
	|	**define** *sn* = *D*;	(Definição Concreta)
	|	**invariant** *D*;	(Invariante de Classe)
field	::=	*T fn* [= *E*]? ;	(Decl. de Variáveis de Instância)
method	::=	*modifier T mn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Métodos)
constructor	::=	*modifier cn*($\overline{arg}$) $\overline{spec}$ { *ST* }	(Declaração de Construtores)
modifier	::=	**public** | **static** | ... | **pure**	(Modificadores)
spec	::=		(Especificação de Procedimentos)
		requires *D*	(Pré-condição)
	|	**ensures** *D*	(Pós-condição)
ST	::=	...	(Comandos)
	|	**assume** *D*	(Assume)
	|	**sassert** *D*	(Assert Estático)

cn $\in$ nomes das classes *sn* $\in$ nomes dos estados

fn $\in$ nomes das variáveis de instância

Exemplo

```
public class Entry {
    private int x;
    invariant x:pos;
    private Entry next;

    public Entry(int x)
        requires x:pos
    { this(x, null); }

    public Entry(int x, Entry n)
        requires x:pos
    {
        this.x = x;
        this.next = n;
    }

    public Entry getNext()
    { return next; }

    public int getElement()
        ensures return:pos
    { return x; }
}
```

```
public class Stack {
    private Entry head;

    public Stack()
        ensures true + head:null
    { head = null; }

    public void push(int i)
        requires i:pos
        ensures true + !head:null
    { head = new Entry(i, head); }

    public int pop()
        requires true + !head:null
        ensures return:pos
    {
        int res = head.getElement();
        head = head.getNext();
        return res;
    }
}
```

Verificação de um Programa

A verificação é efectuada com base num cálculo de pré-condições mais fracas, que também concebemos, baseado na lógica dual.

O algoritmo de verificação de um programa face à sua especificação é composto pelas seguintes fases:

1. Geração de condições de verificação para cada procedimento da classe;
2. Reescrição das propriedades e das condições do Java em predicados da lógica proposicional;
3. Submissão das fórmulas obtidas ao SMT-Solver.

Verificação de um Programa

Cálculo de pré-condições mais fracas:

- Predicados transformadores definidos para cada comando da linguagem;
- Formulação dual (Hoare-Separação) para permitir lidar com *aliasing*.
- As regras do cálculo são aplicadas ao corpo de cada procedimento do fim para o início, a partir da pós-condição.

As regras para o cálculo WP OO estão divididas em dois grupos:

- Comandos Independentes
  (composição seq., ciclos, **assume**, **sassert**, **if**)
- Comandos Puros/Lineares
  (afectação, retorno, instanciação, invocação de métodos, **synchronized**)

Cálculo WP

Pré-condição dual mais fraca de um comando

Sendo ST um comando e $C + S$ uma pós-condição dual, então, a pré-condição mais fraca correspondente é representada na forma:

$$wp(ST, C + S)$$

```
public static void pushOn(Stack s, int i)
    requires i:pos + !s:null
{


    Stack b = s;


    b.push(i);

    sassert true + !b:null;
    ⊤ + ∅
}
```

Cálculo WP

Pré-condição dual mais fraca de um comando

Sendo ST um comando e $C + S$ uma pós-condição dual, então, a pré-condição mais fraca correspondente é representada na forma:

$$wp(ST, C + S)$$

```
public static void pushOn(Stack s, int i)
    requires i:pos + !s:null
{


    Stack b = s;


    b.push(i);
    WP :   ⊤ + ¬Null(b) = C1 + S1
    sassert true + !b:null;
    ⊤ + ∅
}
```

Cálculo WP - Assert Estático

$$wp\,(\textbf{sassert}\ A, R) = A \wedge R$$

Exemplo

b.push(i);

WP: $\top + \neg Null(b) = C_1 + S_1$

$$\frac{\texttt{[invocação linear sem retorno - ml/ol]} \qquad S \downarrow \{\overline{z}\} = \emptyset \qquad \texttt{true} + ((k \neq \texttt{null} \wedge R_{mn_B}[this/k, \overline{p_1}/\overline{y}]) \Rightarrow S \downarrow \{k, \overline{z}\})}{wp\,(k.mn(\overline{y}, \overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k, \overline{p_1}/\overline{y}] \Rightarrow C) \end{array} \right) \\ + \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \\ * \\ S - \{k, \overline{z}\} \end{array} \right) \right) \end{array} \right)}$$

Q_{mn}/R_{mn} – pré/pós-condição dual do método mn

Q_{mn_A}/R_{mn_A} – parte pura $\quad Q_{mn_B}/R_{mn_B}$ – parte linear $\quad I_c$ – invariantes de classe

Exemplo

$WP: \quad > (i, 0) + \neg Null(b)$
$VCs: \quad \top + \neg Null(b) * \neg Null(b.head) \Rightarrow S_1 \downarrow \{b\}$

```
b.push(i);
```

WP: $\quad \top + \neg Null(b) = C_1 + S_1$

[invocação linear sem retorno - ml/ol]

$$\frac{S \downarrow \{\overline{z}\} = \emptyset \qquad \texttt{true} + ((k \neq \texttt{null} \wedge R_{mn_B}[this/k, \overline{p_1}/\overline{y}]) \Rightarrow S \downarrow \{k, \overline{z}\})}{wp\,(k.mn(\overline{y}, \overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k, \overline{p_1}/\overline{y}] \Rightarrow C) \end{array} \right) \\ + \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \end{array} \right) \right) \\ * \\ S - \{k, \overline{z}\} \end{array} \right)}$$

Q_{mn}/R_{mn} – pré/pós-condição dual do método mn

Q_{mn_A}/R_{mn_A} – parte pura $\qquad Q_{mn_B}/R_{mn_B}$ – parte linear $\qquad I_c$ – invariantes de classe

Exemplo

$WP: \quad > (i, 0) + \neg Null(b)$
$VCs: \quad \top + \neg Null(b) * \neg Null(b.head) \Rightarrow S_1 \downarrow \{b\}$ ✓

```
b.push(i);
```

WP: $\quad \top + \neg Null(b) = C_1 + S_1$

[invocação linear sem retorno - ml/ol]

$$\frac{S \downarrow \{\overline{z}\} = \emptyset \qquad \texttt{true} + ((k \neq \texttt{null} \wedge R_{mn_B}[this/k, \overline{p_1}/\overline{y}]) \Rightarrow S \downarrow \{k, \overline{z}\})}{wp\,(k.mn(\overline{y}, \overline{z}), C + S) = I_c[this/k] \Rightarrow \left(\begin{array}{c} \left(\begin{array}{c} Q_{mn_A}[this/k, \overline{p_1}/\overline{y}] \\ \wedge \\ (R_{mn_A}[this/k, \overline{p_1}/\overline{y}] \Rightarrow C) \end{array} \right) \\ + \\ \left(\left(\begin{array}{c} k \neq \texttt{null} \\ \wedge \\ Q_{mn_B}[this/k, \overline{p_1}/\overline{y}, \overline{p_2}/\overline{z}] \end{array} \right) \right) \\ * \\ S - \{k, \overline{z}\} \end{array} \right)}$$

Q_{mn}/R_{mn} – pré/pós-condição dual do método mn

Q_{mn_A}/R_{mn_A} – parte pura $\qquad Q_{mn_B}/R_{mn_B}$ – parte linear $\qquad I_c$ – invariantes de classe

Cálculo WP

Pré-condição dual mais fraca de um comando

Sendo ST um comando e $C + S$ uma pós-condição dual, então, a pré-condição mais fraca correspondente é representada na forma:

$$wp(ST, C + S)$$

```
public static void pushOn(Stack s, int i)
    requires i:pos + !s:null
{


    Stack b = s;
    WP :   >(i,0) + ¬Null(b) = C2 + S2
    VCs :   ⊤ + ¬Null(b) * ¬Null(b.head) ⇒ S1 ↓ {b}      ✓
    b.push(i);
    WP :   ⊤ + ¬Null(b) = C1 + S1
    sassert true + !b:null;
    ⊤ + ∅
}
```

Cálculo WP

Pré-condição dual mais fraca de um comando

Sendo ST um comando e $C + S$ uma pós-condição dual, então, a pré-condição mais fraca correspondente é representada na forma:

$$wp(ST, C + S)$$

```
public static void pushOn(Stack s, int i)
    requires i:pos + !s:null
{
    WP :   >(i,0) + ¬Null(s)
    VCs :   S2 ↓ {s} = ∅
    Stack b = s;
    WP :   >(i,0) + ¬Null(b) = C2 + S2
    VCs :   ⊤ + ¬Null(b) * ¬Null(b.head) ⇒ S1 ↓ {b}      ✓
    b.push(i);
    WP :   ⊤ + ¬Null(b) = C1 + S1
    sassert true + !b:null;
    ⊤ + ∅
}
```

Cálculo WP - Afectação Linear

$$\frac{y \neq \texttt{null} \quad x \neq y \rightarrow S \downarrow \{y\} = \emptyset}{wp\,(x = y, C + S) = C + S[x/y]}$$

Cálculo WP

Pré-condição dual mais fraca de um comando

Sendo ST um comando e $C + S$ uma pós-condição dual, então, a pré-condição mais fraca correspondente é representada na forma:

$$wp(ST, C + S)$$

```
public static void pushOn(Stack s, int i)
    requires i:pos + !s:null
{
    WP :  >(i,0) + ¬Null(s)
    VCs :  S2 ↓ {s} = ∅   ✓
    Stack b = s;
    WP :  >(i,0) + ¬Null(b) = C2 + S2
    VCs :  ⊤ + ¬Null(b) * ¬Null(b.head) ⇒ S1 ↓ {b}   ✓
    b.push(i);
    WP :  ⊤ + ¬Null(b) = C1 + S1
    sassert true + !b:null;
    ⊤ + ∅
}
```

Cálculo WP - Afectação Linear

$$\frac{y \neq \texttt{null} \quad x \neq y \rightarrow S \downarrow \{y\} = \emptyset}{wp\,(x = y, C + S) = C + S[x/y]}$$

Cálculo WP

Pré-condição dual mais fraca de um comando

Sendo ST um comando e $C + S$ uma pós-condição dual, então, a pré-condição mais fraca correspondente é representada na forma:

$$wp(ST, C + S)$$

$\Big(> (i, 0) + \neg Null(s) \Big) \Rightarrow \Big(> (i, 0) + \neg Null(s) \Big)$

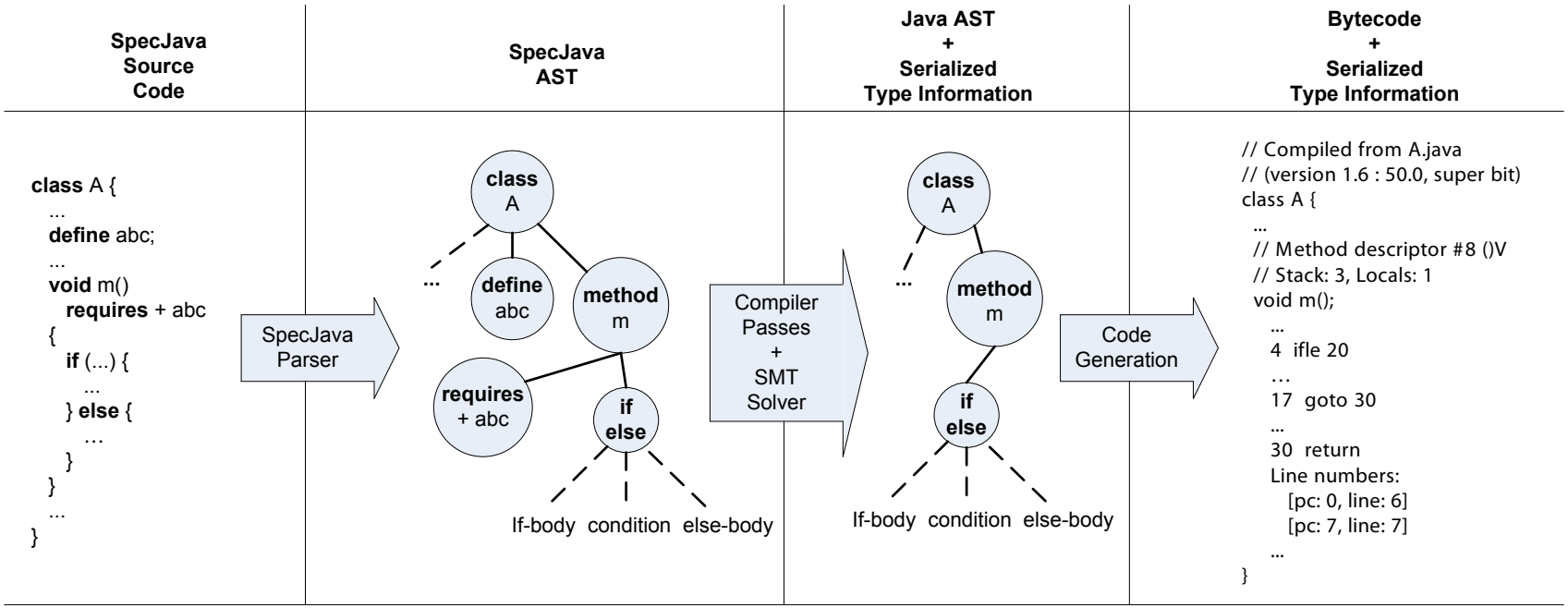
```
public static void pushOn(Stack s, int i)
    requires i:pos + !s:null
{
    WP :  > (i, 0) + ¬Null(s)
    VCs :  S2 ↓ {s} = ∅   ✓
    Stack b = s;
    WP :  > (i, 0) + ¬Null(b) = C2 + S2
    VCs :  ⊤ + ¬Null(b) * ¬Null(b.head) ⇒ S1 ↓ {b}   ✓
    b.push(i);
    WP :  ⊤ + ¬Null(b) = C1 + S1
    sassert true + !b:null;
    ⊤ + ∅
}
```

SpecJava

DEMO

Implementação

A extensão da linguagem Java foi implementada recorrendo à ferramenta Polyglot, que implementa um compilador extensível para a linguagem Java. [Nathaniel Nystrom, Michael Clarkson e Andrew Myers] (foram produzidas aprox. 11500 linhas de código).



http://cs.nyu.edu/acsys/cvc3/

http://www.cs.cornell.edu/Projects/polyglot/

http://ctp.di.fct.unl.pt/~tsantos/tools/specjava.zip

Trabalho Futuro

- A Lógica Dual é uma aproximação nova
  - Requer um estudo das suas propriedades
- O cálculo WP OO não foca alguns aspectos do Java, que seriam interessantes abranger
  - Mais instruções de controle, excepções
  - Propriedades de forma de estruturas de dados
  - Herança
- Actualmente o protótipo suporta especificações de classes criadas pelo programador, seria interessante suportar também
  - Especificações de Interface
  - Especificações das Bibliotecas do Java